MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LINCOLN LABORATORY

THE MESH SYNCHRONOUS PROCESSOR
MeshSP™

I.H. GILBERT
W.S. FARMER
Group 46

TECHNICAL REPORT 1004

14 DECEMBER 1994

Approved for public release; distribution is unlimited.

LEXINGTON MASSACHUSETTS

EXECUTIVE SUMMARY

INTRODUCTION AND SCOPE

The Mesh Synchronous Processor (MeshSP) is a parallel computer architecture well suited to a broad range of computationally intensive problems. For a given level of performance, hardware realizations of the MeshSP are unusually economical, low-powered, and compact. The MeshSP presents an architectural simplicity which is unique in the field of high-performance parallel processing systems and facilitates software development. The architecture is multiuse, highly extensible, and will incorporate new device technology while maintaining investments in software and algorithm development.

The MeshSP is a single instruction-stream multiple data-stream (SIMD) architecture comprising an array of processors connected via a two- or three-dimensional nearest-neighbor mesh. Its current realization incorporates a single monolithic processor element (PE), the Analog Devices' ADSP-21060 SHARC IC. This commercial integrated circuit was specifically designed to meet the requirements of a MeshSP PE. This chip provides the features required for digital signal-processing (DSP) environments, contains a large memory, and incorporates special hardware to facilitate the SIMD-compatible data transfers.

These developments are the result of a 12-year program of SIMD processor research. Two antecedent processors, the SP-1 (1984) and SP-2 (1986), were developed to demonstrate various concepts related to highly parallel synchronized processing and algorithm parallelization. The emphasis has been on processing efficiency, hardware simplicity, and the application of SIMD techniques to a widening range of applications.

The scope of this document is to

1. Define the MeshSP architecture
2. Describe features of the SHARC chip directly concerned with the architecture
3. Specify the required programming considerations
4. Describe the existing library of signal-processing operations
5. Describe a MeshSP simulation tool
6. Provide a collection of sample applications

ARCHITECTURE

The MeshSP includes an array of PEs, a master processor, an input/output (I/O) module, and a host computer. The master is identical to a PE, except that it has direct access to a large off-chip program memory. The program is executed by the master, and the instructions are broadcast to the processor array for parallel execution. Data may be passed between the PE array and the host computer "in the background."

The host serves as interface to the external world. The PEs incorporate advanced hardware links for interprocessor communication.

Although the emphasis is on SIMD processing, a number of non-SIMD capabilities are supported. The hardware supports multiple instruction-stream multiple data-stream (MIMD) computations when code segments are loaded into PE on-chip memory. Other techniques facilitate conditional code and conditional debugging in a SIMD context.

Data transfers are specified with a standard data structure called the transfer control block (TCB), which is stored in memory. Transfers include both I/O with the host computer and interprocessor data movement. TCBs designate a general arrangement of data as two-dimensional arrays which may be partitions of larger arrays or sparsely sampled arrays. SHARC hardware controls data transfers specifed by a chain of TCBs without intervention by or interference with the core (arithmetic) functions. The unusual capabilities of the SHARC enhance throughput by means of concurrent computation and communication.

Interprocessor communication hardware includes the capability to remap logical directions in the software to physical links. This enables various processor-dependent communication patterns and processor connectivity changes with minimum impact on application software.

The first realization of the architecture is the MeshSP-1, an 8×8 array of toroidally connected PEs. Some MeshSP-1 characteristics are

1. 7.7-Gflops peak throughput
2. 32 Mbytes of PE internal memory
3. 40-Mbytes/sec interprocessor communication rate (per PE)
4. 40-Mbytes/sec I/O rate with the host
5. 100-W (approximately) power requirement
6. 2 circuit boards, 7.25 × 13 in, occupying 0.15 ft^3
7. Two-dimensional torroidal connectivity with fault tolerance

SOFTWARE

MeshSP programs require explict and symmetric problem decomposition across the PEs. This is the responsibility of the algorithm designer; there are no parallelizing tools. Although much recent research addresses the issue of automatic problem parallelization, the MeshSP is intended for demanding applications where the investment of effort to design a parallel algorithm is justified.

The application software is written in ANSI standard C with no parallel extensions. SIMD considerations do impose programming restrictions, which are enforced by a few clearly stated programming

rules. We distinguish three classes of variables, depending on their location (on-chip or off-chip) and whether they are multiple-valued across PEs.

A substantial library of callable routines is available to specify and direct data transfers. These range from the lowest level, specification of the contents of TCBs, to very high level communication patterns spanning many (or all) of the PEs. This report includes a detailed exposition of these routines and their use in typical applications.

FUNCTIONAL SIMULATOR

The MeshSP functional simulator permits the execution of MeshSP application code on an inexpensive workstation without the need for MeshSP hardware. Simulation permits the development, testing, and debugging of MeshSP applications written in C. A multitasking operating system is employed to represent program execution in each PE separately.

ALGORITHMIC EXAMPLES

We provide a collection of diverse algorithms to illustrate the process of parallel decomposition on the MeshSP. The examples include the elementary problem of averaging data dispersed across PEs; the important signal-processing problem of computing large Fourier transforms; and some advanced applications involving linear algebra, neural networks, and tomographic reconstruction.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (Continued)

LIST OF TABLES

1. INTRODUCTION

1.1 SYSTEM OVERVIEW

The Mesh Synchronous Processor (MeshSP) is a parallel processor architecture providing an economical solution for computationally demanding multidimensional signal-processing problems. It is also suitable for applications such as three-dimensional graphics, neural networks (multilayer perceptrons), tomographic reconstruction, and the solution of large systems of linear equations. The MeshSP operates primarily as a single instruction-stream, multiple data-stream (SIMD) processor with nearest-neighbor mesh communications. A consequence of this architectural simplicity is that the MeshSP appears to the programmer as a single computer that executes a single program.

A commercial integrated circuit, the Analog Devices' SHARC (ADSP-21060), has been developed to meet all requirements of a MeshSP processor element (PE). The SHARC is not limited to this role, but it may be used in a variety of other multiprocessor and uniprocessor configurations. Some of the SHARC's key features are

1. 120-Mflops (peak) throughput
2. Nested zero overhead loops and delayed branching
3. Instruction set compatible with SIMD operation (data insensitive)
4. 512 Kbyte of fast, on-chip memory (SRAM)
5. Noninterfering access of memory by the processor core and communication systems
6. Six communication ports, each having a 40-Mbyte/sec bandwidth
7. 5-Mbyte/sec input/output (I/O)
8. 16-, 32-, and 40-bit floating-point formats
9. Two-dimensional DMA controllers

The first realization of the MeshSP architecture, MeshSP-1, consists of an array of 64 PEs (also termed **slaves**) arranged on an 8×8 rectangular grid. It consists of two boards in an IBM-compatible 486 personal computer, consuming about 100 W of power and providing a peak throughput of 7.7 Gflops, rivaling that of supercomputers (see Figure 1-1). These boards were designed and fabricated by Avid Technologies Inc., of Twinsburg, Ohio.

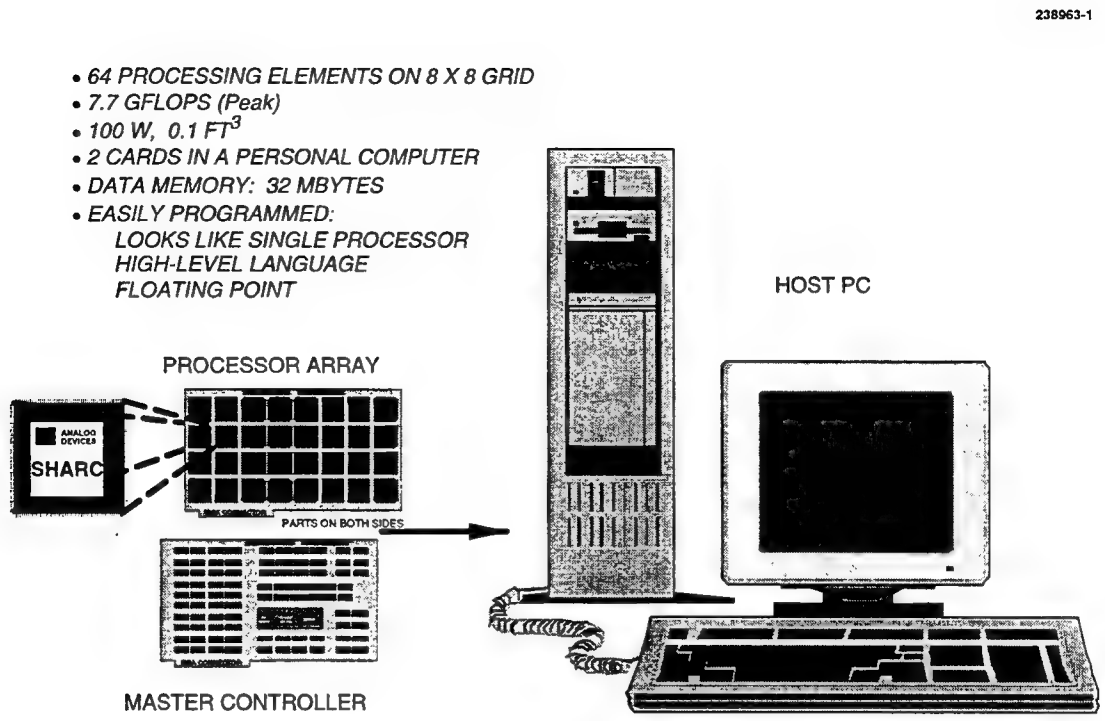

Figure 1-1. MeshSP-1 System.

The MeshSP architecture is highly extensible. The SHARC chip was designed to provide considerable tolerance to interprocessor timing skew and delay, permitting the construction of arrays of thousands of elements without sacrificing reliability. In addition, various mesh connectivities are feasible: two- and three-dimensional rectangular meshes, as well as two-dimensional triangular and hexagonal meshes. The principal restriction is that each SHARC can be connected to no more than six other SHARCs.

1.2 SCOPE OF THIS DOCUMENT

This document provides a comprehensive description of the MeshSP architecture, its origin, and its application to a number of representative problems. It is intended both as a reference for users of the MeshSP and a resource for readers considering the MeshSP as a candidate solution for their computational problem. The emphasis is on those aspects developed at, or specified by, Lincoln Laboratory. For example, details of the SHARC chip unrelated to the specifics of the MeshSP architecture are omitted. They are available in the Analog Devices Users' Manual for this chip. Similarly omitted is documentation related to the commercial (non-Lincoln) software tools used to run and debug MeshSP programs.

1.3 BACKGROUND AND DESIGN PHILOSOPHY

In 1982 Lincoln Laboratory began a study of computer architectures for real-time, multidimensional signal processing. The study involved paper designs and the eventual construction of some small hardware demonstration systems. The architectures were oriented toward potential applications for which considerations of size, weight, power, performance, maintainability, programming flexibility, and cost were important. It was assumed that the processing task far exceeded the capabilities of any available uniprocessor and that a multiprocessor solution was required.

The usual response to such demanding processing requirements is a specialized design in which separate processing functions are allocated to separate hardware subsystems. This approach was rejected as being costly and inflexible. Such organization makes it difficult to reallocate processor resources between hardware subsystems as requirements change. In addition, the unique hardware of each subsystem demands its own design effort.

1.3.1 Seamless Processing

A design of seamless and uniform fabric of processors was chosen over which the data could be distributed. These processors would be fully programmable, thereby allowing arbitrary redistribution of the various processing tasks in time. Because neither processing power nor memory would be allocated to specific subsystems, these resources could be reallocated to different tasks without restriction. This approach achieves the computational power of a multiprocessor while approximating the flexibility and simplicity of a uniprocessor.

1.3.2 Synchronized Processing

Another major architectural issue concerns coordination of activities in the individual PEs. One possible approach is multiple instruction-stream, multiple data-stream (MIMD) or independent operation. Here, each processor has its own code as well as data and is free to execute a conventional program with arbitrary data dependence. An alternative is SIMD or lock-step operation, where the entire array of slaves executes instructions broadcast by a single master. SIMD operation allows a large fraction of hardware and time-line resources to be devoted to pure computation, minimizing the portion devoted to such unproductive activities as control and handshaking. Furthermore, because the slaves need no copies of the code, their local memories can be efficiently devoted to data while the single copy of code at the master can be optimized for speed rather than size. These considerations of efficiency and throughput led us to choose SIMD control, which is a choice now confirmed by extensive experience.

1.3.3 Nearest-Neighbor Mesh Communications

The final architectural choice concerns interprocessor connectivity. In comparing the various alternatives it is important to distinguish between two performance measures: bandwidth and latency. High bandwidth (total words/second) is ensured by providing a sufficient number of parallel communication paths

running at adequate speeds. Low latency (maximum delay from transmission to reception) requires minimizing the maximum path length. For example, the 12-dimensional hypercube of the original CM-1 Connection Machine allows data to be passed from any of the 4096 elements to any other in at most 12 basic steps. On the other hand, an equal number of elements arranged as a 64×64, toroidal, nearest-neighbor mesh requires a maximum of 64 basic steps for the most remote transfer. Most of the applications that were examined were dominated by local transfers and thus low latency. Furthermore, computation and communication can often be pipelined so that the required data are in place when needed. Considerations of simplicity, hardware resources, and freedom from conflict and contention led to the choice of the nearest-neighbor, toroidally connected, rectangular mesh.

1.3.4 SP-1

Our first parallel Synchronous Processor (SP-1) was designed to explore these concepts and design choices using such chips as were commercially available in 1982. It comprises an array of 16 PEs, each containing an Intel 8086 microprocessor and 8087 numeric coprocessor, and 64 Kbytes of local memory. The master controller of SP-1 was identical to the slave PEs, only differing in that it was provided program memory as well as data memory. As the master executed its program, the instructions it fetched from its program memory were broadcast to the array of slaves.

These early Intel microprocessors were not intended for synchronous operation, as the execution times for various instructions (such as multiply or add) were data dependent. SIMD operation therefore required determining when all slaves completed each operation, and only then letting the master proceed to the next instruction.

SP-1 was completed in 1984. During the intervening period, a substantial number of SIMD compatible algorithms was devised, and the basic soundness of the architecture was validated. However, the performance of SP-1 was limited by that of its components. In aggregate, the SP-1 performance was only 0.5 Mflops.

1.3.5 SP-2

By 1984 a new class of commercial components became available, higher-performance integer digital signal-processing (DSP) chips. In particular, the Texas Instruments TMS32020 allowed single-cycle computation at a clock frequency of 6 MHz (followed soon thereafter by the 10-MHz TMS320C25). SP-2 was constructed as an array of 64 TMS32020s. The TMS32020 had a modest amount of on-chip memory, 544 16-bit words, which was was augmented by 64K 16-bit words of slower (DRAM) off-chip memory for each slave.

To maximize performance, a more complex control strategy was employed–one that provided concurrent control, arithmetic, interprocessor communication, and I/O. The first element of concurrency was provided by writing separate and independent programs for computation, communication, and I/O. These three programs were then executed by three separate programmable controllers: the array master for computation and control, the communication master for interprocessor communication, and the host for I/O (Figure 1-2).

The programmer was responsible for coding each function for the appropriate controller and establishing the necessary synchronization points.

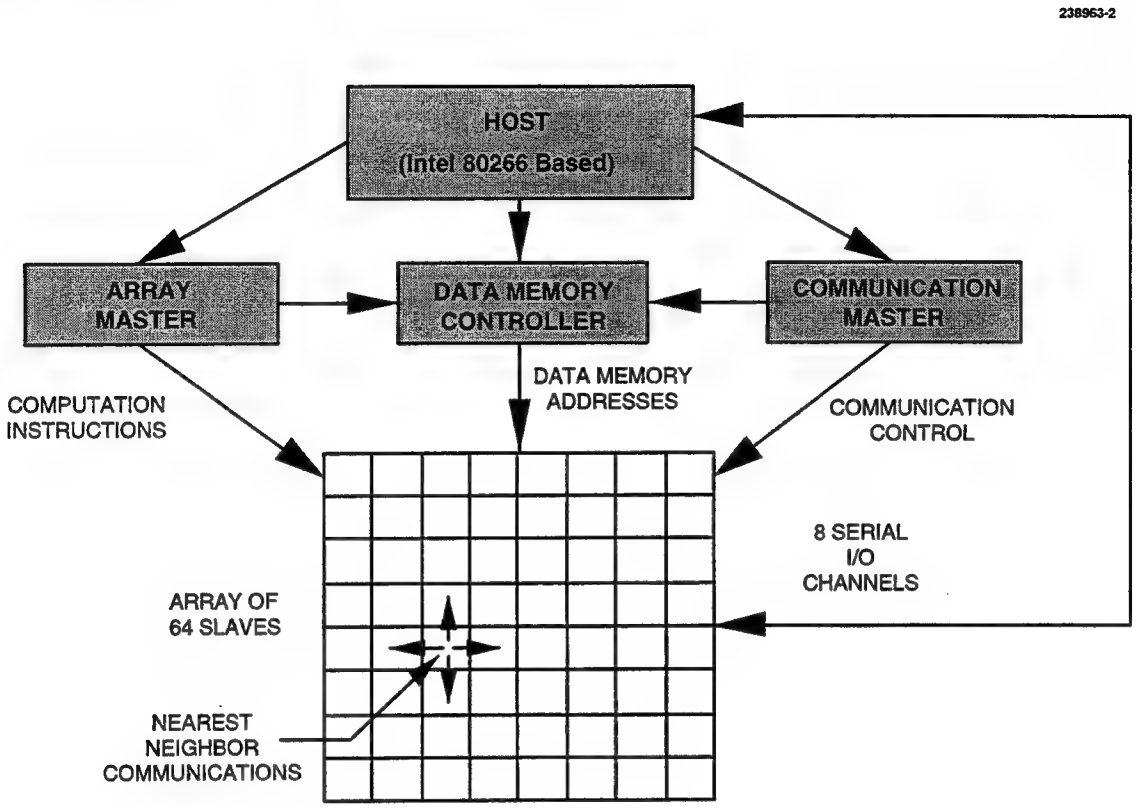

Figure 1-2. SP-2 architecture (circa 1986).

1.3.6 Separation of Computation and Control

Another innovation was introduced to further increase run-time computational efficiency. As a conventional microprocessor, the TMS32020 was intended to execute code in which control and computation were sequentially interspersed. In a tight inner loop, for example, a significant part of the time line might be devoted to checking a loop counter and branching back if the counter had not decremented to zero. This control activity would have been duplicated wastefully in every PE and frustrated the efforts at filling the time line with useful arithmetic. This inefficiency was overcome by providing the array master controller with the means to access two blocks of code: one containing control instructions for its own use alone, a second containing arithmetic instruction broadcast to the slaves. By proper buffering of these operations it was possible to keep the stream of purely arithmetic instruction to the slaves fully filled. This strategy was facilitated by development of a unique code translator which allowed the programmer to write conventional code in which control and computation were interspersed. This software tool then separated the code into its two components in preparation for downloading to the two code spaces in the array master.

SP-2 was operational in 1986 with Texas Instruments 6-MHz TMS32020 chips, and later upgraded to 10-MHz TMS320C25s with 256K words of off-chip memory. In its later form it demonstrated a performance of 640 MOPS (16-bit integer operations, 32-bit intermediate precision), which was more than three orders of magnitude faster than SP-1. SP-2 is described in more detail in The Lincoln Laboratory Journal [1].

1.3.7 Development of a Monolithic PE

By 1990, rapid improvements in DSP technology opened the way for substantial improvements in Synchronous Processor (SP) systems. It now became realistic to consider implementing a complete high-performance SP slave element as a single integrated circuit, which permits construction of systems that meet stringent requirements in terms of throughput, size, weight, and power. If such a chip could be manufactured and sold on a large scale, it would offer the least expensive route to future SPs. However, there was no reason to believe that a suitable chip would come into existence unless Lincoln Laboratory played a major role in its definition and development. In 1990 features to be incorporated into such a chip so as to optimize it as a PE for the SP were explored.

The new monolithic slave would be required to provide at least as much off-chip memory as the final form of SP-2, 4 Mbits (there would be no off-chip memory). By the mid-1990s it would be possible to integrate such a memory together with a high-performance DSP core processor on a single chip. While larger (16-Mbit) DRAM memories were available by 1993, the need for a memory cell compatible with logic circuits, and fast enough to incur no wait states, forced the choice of lower density SRAM.

The next major requirement was for a design that would combine the concurrency and computational efficiency of SP-2 with the programming simplicity of SP-1. To some degree, this had been anticipated by developments in the industry. Newer high performance chips such as the TI TMS320C30 now provided such features as zero overhead looping. That is, the on-chip address generators were designed to operate concurrently with computation, which filled the time line with pure arithmetic. The need for separating computation and control (which motivated the SP-2 array master design) had disappeared. However, the requirement for concurrent interprocessor communication and I/O remained. Such concurrency must not sacrifice coding simplicity.

The most important driver in the specification for the new monolithic slave element was the need for programming simplicity and software productivity. The widespread availability of high-quality compilers for the C language, together with floating-point arithmetic, overcame many of the deficiencies of the SP-2. Programming would be simpler still if the machine appeared to the programmer as a single processor rather than the triad (array master, communication master, and host) of SP-2. The new system was named the "Monolithic Synchronous Processor," which alluded to both the unity of the programming model and the fabrication of the PE as a monolithic integrated circuit. More recently, the designation MeshSP was adopted, which refers to both the SIMD operation and mesh connectivity.

1.3.8 Concurrent Communications and I/O

The following strategy was chosen to provide communication and I/O concurrency without compromising the unitary nature of programming model: hardware resources would be provided on-chip to support these functions while computation proceeded. These subsystems would be initialized by the core processor but then would run to completion in the background, interrupting the core processor only when the desired transfer was complete. In order not to excessively burden the processor core it would be necessary to equip the communication and I/O subsystems with sufficient intelligence to complete complex transfers with a simple setup and no further assistance. These objectives were then refined to form a set of detailed specifications for the proposed MeshSP slave element.

The MeshSP was intended to be a significant improvement over the SP-2 in all respects (except for memory size, which was constrained by technology). A clock speed of 30 MHz was anticipated, which is about twice as fast as the highest performance floating-point DSP chip then available (the TI TMS320C30), and would require all arithmetic operations to execute in a single cycle. The inter-processor communication frequency was to be tied to this clock as well, increasing from 10 MHz for SP-2 to 30 MHz. In addition, the single bit-wide SP-2 communication link was to be replaced by a nibble-wide link (4 bits), for a total factor of 12 in increased communication speed. Although the clock speed of the serial I/O (or SIO) system would increase to 30 MHz as well, there was no need to widen it as the bandwidth could always be increased via the wiring pattern (as discussed later in the section on the MeshSP SIO system).

1.3.9 MSPSE Development Contract

Having developed a set of detailed specifications for the proposed monolithic Synchronous Processor slave element (MSPSE), in June of 1990 a contract was let for the development of this chip and the procurement of a sufficient number to construct an 8×8 processor demonstration system. A contract was subsequently awarded to Westinghouse Electronic Systems (WEC), the sole bidder to propose the full 4-Mbit on-chip memory. WEC was originally the prime contractor, who then subcontracted the processor development to SGS Thompson (INMOS), the developers of the Transputer and the memory development to INOVA, a small manufacturer of wafer-scale integrated circuits. In addition to negotiating these subcontracts, WEC provided invaluable technical support and systems analysis. By July 1991 it became clear that the new T9000 transputer proposed by INMOS as the basis of the MSPSE was fundamentally incompatible with SIMD operation. In addition, INOVA was purchased by Cypress Semiconductors. The new owner withdrew from further involvement in the MSPSE project.

By the fall of 1991, as we prepared to rebid the MSPSE development contract, WEC identified alternative subcontractors, Analog Devices Inc. (ADI) for the core processor and Electronic Designs Inc. (EDI) for the memory. The actual chip fabrication was to be done at Sharp Corporation in Japan. By March of 1992 the project was reorganized with Lincoln Laboratory serving as the prime contractor and ADI (processor), EDI (memory), and WEC (technical support) acting as subcontractors. In the fall of 1993 the

memory design team left EDI to form a new company I-Chips, Inc. They remained a member of the MSPSE team, completing the memory design.

The addition of ADI to the MSPSE team was extremely beneficial. ADI had just completed their first high-performance floating-point chip, the ADSP-21020. This chip operated at 33 MHz, executing two (and in one case three) arithmetic operations in a single clock cycle. The execution time was independent of the data, qualifying it for SIMD operation. In addition, the ADSP-21020 had neither on-chip memory (beyond a simple instruction cache) nor significant I/O capability. The absence of these features allowed the augmentation of the ADSP-21020 processor core with communication links and a memory structure designed explicitly for MeshSP operation. The result was a commercial PE almost perfectly suited to the SP.

2. MeshSP ARCHITECTURE AND HARDWARE

The MeshSP architecture, shown in Figure 2-1, is quite similar to that of the original Synchronous Processor SP-1. There is a single **master** processor that broadcasts instructions to an array of slave processors. The slaves may execute such broadcast code, or they may execute code from their internal memories. The master processor is essentially identical to a slave, differing solely in that the master alone has access to a large external memory. This memory is used primarily to store the program, as well as data used jointly by the master and all slaves. The master may be interrupted by a signal that is either AND-ed or OR-ed across all slaves. This allows the master to relinquish and reestablish SIMD processing, and to respond flexibly to various error conditions.

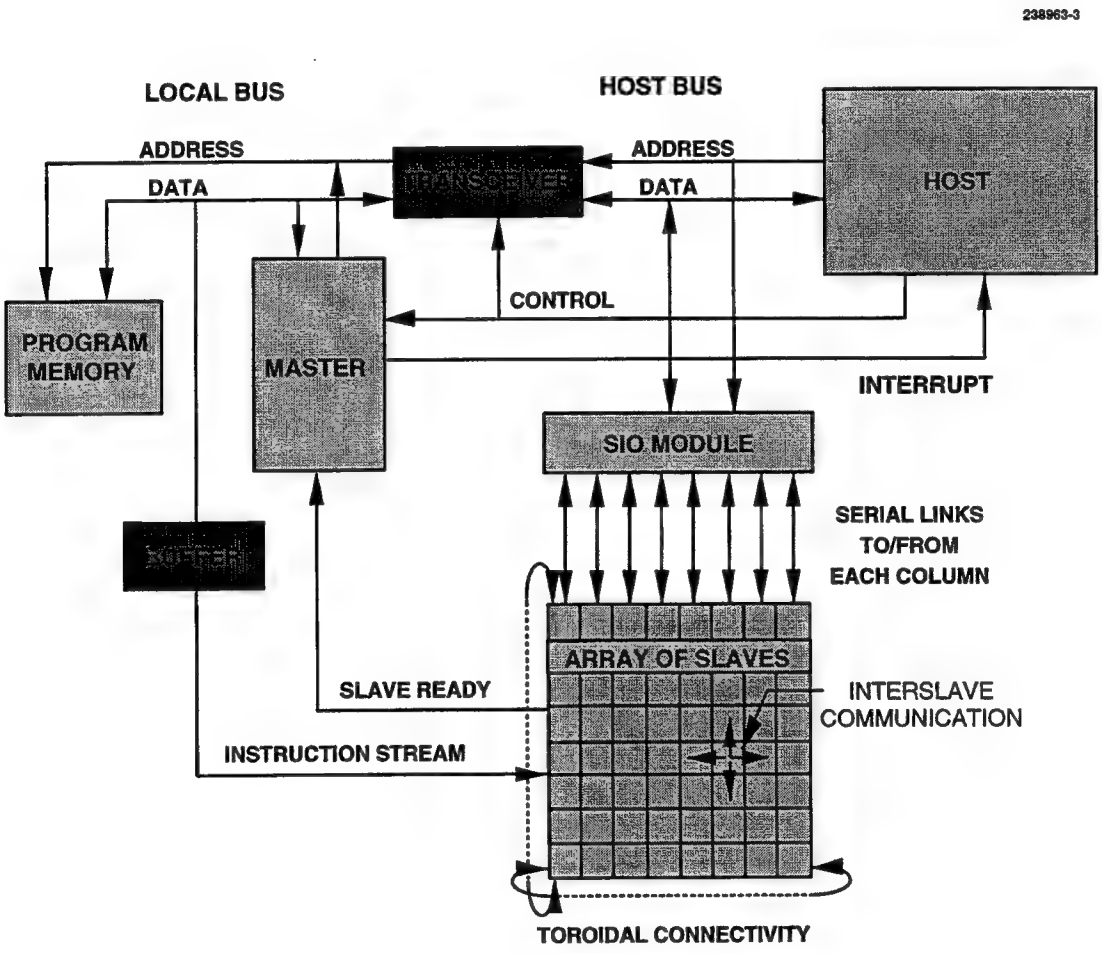

Figure 2-1. MeshSP architecture.

The **host** is a conventional computer that provides overall control of the MeshSP as well as communication with the outside world. It has access to the master's external memory via a transceiver. The host may initialize and interrupt the master, and it may be interrupted by the master to provide services.

The **Serial Input/Output (SIO)** system consists of the SIO module (SIOM), a custom integrated circuit, plus a number of bit-serial links connecting the SIOM to the array. It provides the means for delivering multiple-valued data to and from the array.

The array also has high-speed interslave **links**. Figure 2-1 indicates nearest-neighbor communications on a two-dimensional toroidal topology. There are six link ports per slave, thereby making a number of other topologies possible.

2.1 SIMD PROCESSING

The MeshSP normally operates in SIMD mode. The master executes the program from its external memory. As each instruction is fetched, it appears on the data bus and is broadcast to the array of slaves. Each slave (being a complete microcomputer) attempts to fetch its own instructions by generating addresses into its own (nonexistent) external memory space. Each broadcast instruction from the master arrives at the slave in time to satisfy the slave memory read. Thus, each slave effectively has a copy of the master's program. The same procedure can provide the slaves with copies of data items in the external memory.

SIMD operation depends on the proper timing for the receipt of data from the master. Distribution of the broadcast information involves electrical buffering and transmission delays, particularly for large slave arrays. Synchronization is ensured by broadcasting the system clock along with the instruction bus. Thus, slave processing may follow master processing by several clock cycles. The delays are arranged so that the slaves are closely synchronized.

2.2 MIMD OPTIONS

The slave capability to execute code from internal memory enables limited forms of MIMD processing. The master, executing its SIMD program, may broadcast a section of code (as data) to the internal memory space. The code must be self-contained in terms of program flow; no jumps out of the section are allowed. The master may transfer control to this internal code, and all the slaves will do the same. Once internal code execution is under way, the usual SIMD restrictions are not in effect. For example, the slaves may independently branch on the basis of slave-dependent data values. But there must be a mechanism for reestablishing SIMD operation.

2.2.1 Full MIMD Operation

SIMD synchronization can be regained with the following handshaking dialogue. As MIMD processing commences, the master and slaves set their slave-ready output signals to FALSE. The master slave-ready output is used to generate an external signal that inhibits slave operation if an external memory access is made. As the slaves independently finish their routines, they set their slave-ready line to TRUE and branch

to a prearranged location in external memory, which causes a pause. Meanwhile, the master polls its slave-ready input signal which becomes true only when all the slaves are simultaneously ready. Then the master sets its slave-ready output to TRUE, releasing the slaves for subsequent SIMD processing.

The MIMD mechanism permits arbitrarily many MIMD code segments to be executed, with a broadcast of the code and a reestablishment of SIMD operation for each segment. The limited amount of on-chip memory constrains the size of these MIMD code segments.

2.2.2 Simple MIMD Conditionals

The system also supports a simpler "if-then-else" construct in an MIMD context. The code for the "then" clause is maintained in external memory, while the code for the "else" clause is moved to internal memory. The code is structured so that

1. The "else" clause is written as a jump to internal code, followed by a jump back to the location following the "then" clause.
2. The two clauses take the same number of cycles to execute (with NOP padding, if necessary).
3. The condition is formulated to be always true for the master.

The slaves that branch to internal memory cease listening to the master's broadcast "then" clause. The equality of execution time brings all slaves back into SIMD operation without any handshaking.

2.2.3 Conditional Debugging

The slave-ready signal can also be used to support slave-dependent debugging. An error condition is often anticipated at a particular point in the processing stream, such as a call to the square root function with a parameter outside the legal domain. A code fragment may be inserted at that point to check for that error condition. If the condition is violated in any slave, that slave sets the slave-ready signal to FALSE, interrupting the master. The master may then output a message to the user screen. It is necessary to compensate for the delay in the master's response by the insertion of a few NOPs in the code fragment.

2.3 DATA STORAGE AND SIMD OPERATION

During SIMD operation the MeshSP program may access both external (master) memory and internal (master and slave) memory. Variables stored in external memory are termed **master** variables. When read by the program, the value of a master variable is fetched by the master and broadcast to the slaves. When a master variable is written, it is written to external memory by the master alone. Because a master variable exists as a single copy, it may be freely used in conjunction with data dependent transfers of control: if statements, loop indices, subroutine calls, etc.

Variables stored internally must be treated more carefully since their values in the master and the various slaves are not necessarily tied together. To maintain proper SIMD synchronization, two classes of internal variables are distinguished: single-valued (or simply **single**) variables which are forced by the program to be identical in the master and all slaves and multivalued (or simply **multi**) variables whose values may differ between slaves. Single variables exist as multiple physical copies with a single numerical value, allowing them to used as freely as master variables.

Single variables consume valuable internal memory. They are used in place of master variables primarily in two cases. The run-time program stack is necessarily maintained in internal memory to accommodate slave-dependent parameters. Because all variables declared within a function are maintained on the stack, they are are stored in internal memory. When such variables are used for control transfer, they must be single variables. In addition, it is often useful to maintain variables internally so as to free the instruction bus. This allows internal data and external instructions, or internal and external data, to be made available to the core processor on a single cycle.

It is important that the program maintain the proper synchronization of single-valued variables and not make inappropriate use of multivalued variables. This is ensured by following the programming rules presented in Section 3.1

2.4 MASTER-HOST INTERFACE

To the programmer the MeshSP is a conventional computer with all the usual facilities for storing and accessing data and communicating with the outside world. This has been accomplished without providing the MeshSP with its own operating system, but by exploiting the very capable operating system of the host.

All MeshSP operating system requests are initiated directly from the (single) MeshSP program. The usual C language calls for such services as opening files, reading the keyboard, etc., have been replaced in the MeshSP library with software which does not directly provide the desired service but acts indirectly through the host operating system.

To invoke any host operating system service, the master first writes a word identifying the desired service into a specific location in the external master memory. The master then copies the parameter list of the calling function to a known location in the external master memory. These parameters are usually pointers to data in master but may, in some cases, be data themselves. The host is then interrupted and branches to a general-purpose master-host interrupt service routine.

The first action of this interrupt service routine is to cause the master to relinquish the bus to its external data memory by means of the external bus tristate (TS) signal. The same signal is broadcast to the array to maintain synchronism between the master and slaves. At this point the host is free to access this memory without interference from the master. The host then reads the requested service identifier and parameter list from master memory. Based on that information, the host performs the requested service by transferring data between host peripherals (keyboard, screen, disk files, etc.) and external memory. Finally, the host zeroes

the service identifier in shared memory to indicate completion. At that point the TS signal is released, which allows array processing to resume. The master, resuming activity, finds the service identifier to have been reset and resumes processing. Depending on the nature of the requested service, the host may read or write various other locations in external memory.

This same mechanism is used to initialize the system. Here, both the TS and reset signals are first asserted, holding the MeshSP master at reset. The host then downloads code and data to the external program memory and then releases the master to begin processing.

2.5 AUTONOMOUS COMMUNICATION AND I/O VIA DATA STRUCTURES

The MeshSP has been designed to carry out extensive and complex data transfers with minimal impact on computation. The two types of data transfers are interprocessor communication (denoted **CM**) and transfers between the host and the slave array via the Serial I/O links (denoted **SIO**). Both may proceed concurrently with computation, and both are specified by compact data structures in memory.

CM and SIO transfer specification is done with structures called **transfer control blocks (TCBs)**. TCBs may be stored in external master memory if the transfers have no slave dependency. Often this is not the case, and the TCBs must be stored internally. If memory must be conserved, the TCBs may be created "on the fly." If processor time is at a premium, the TCBs may be created once and then repeatedly reused. Often, a mixed strategy is appropriate.

CM and SIO data transfers are carried out by direct memory access (DMA) hardware which is independent of, and noninterfering with, the computational core processor. Two DMA channels are provided for CM and two more for SIO. In each case one channel (receive) serves to move data into the slave, while the other channel (transmit) moves data from the slave. In the case of slave-to-slave communication, the need for two channels is obvious as each slave is necessarily both a source of data and a destination. Two channels are required for SIO as well because the SIO system is not only capable of simultaneous input and output, but it is actually incapable of operating in one direction alone. One-way transfers are performed by transferring dummy data to or from a fixed internal memory location.

Corresponding to the receive and transmit channels for CM or SIO, each TCB consists of two data blocks, one for each channel. Although the SHARC chip does not assume any relation between the locations of these two portions, the MeshSP software does. As shown in Figure 2-2, a MeshSP TCB is a single data structure, including a transmit section and a receive section.

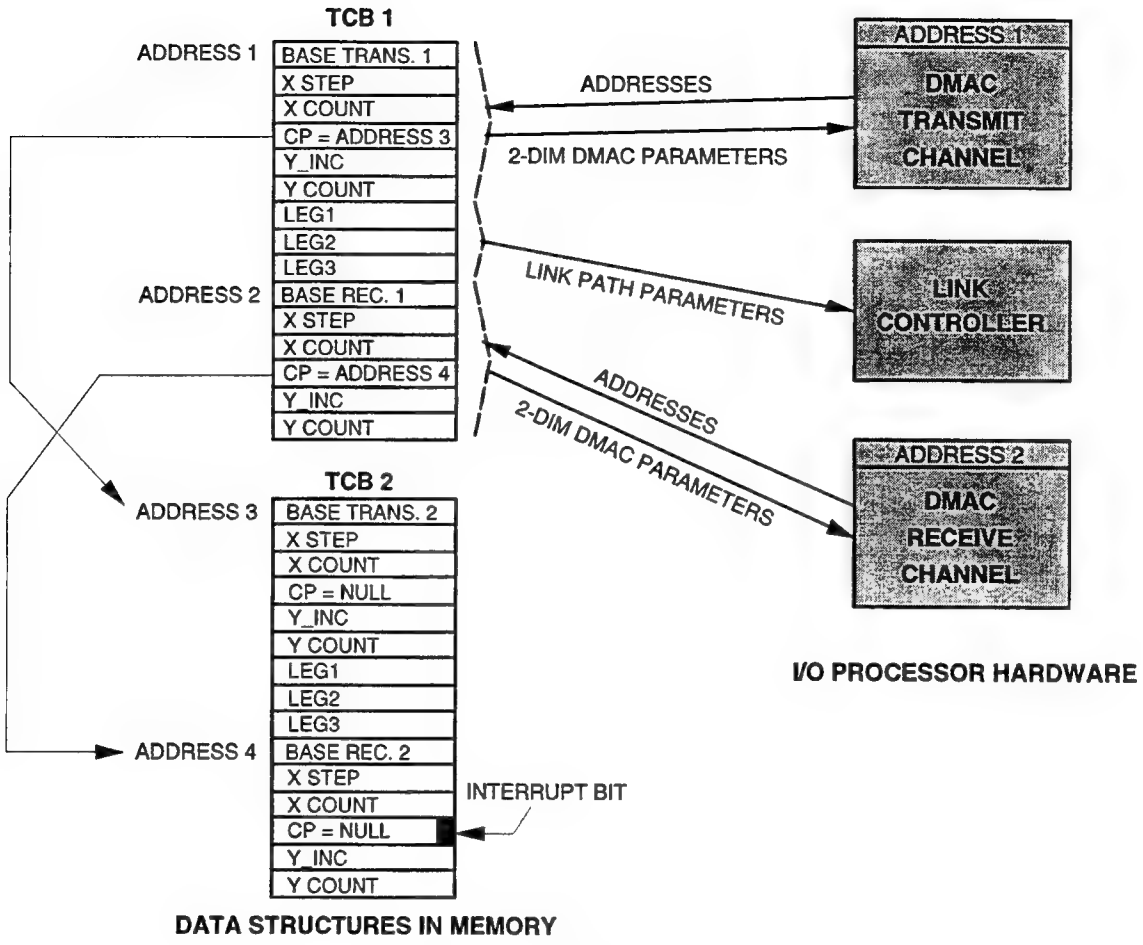

Figure 2-2. TCBs and chaining.

2.5.1 Auto-chaining

Each DMA channel contains a register (chain pointer) which may be initialized with the starting location of the send or receive portion of a TCB. When these registers hold zeroes the transfer hardware is quiescent. To initiate a CM or SIO transfer, the core processor loads the transmit chain-pointer register with the starting address of the transmit portion of the TCB, and then loads the receive chain-pointer register with the starting address of the receive portion of the TCB. At that point the transfer hardware becomes active, and the core processor is free to turn its attention to other things.

The first thing the CM or SIO processor does is to transfer the contents of transmit and receive portions of the TCB to internal control registers. These registers determine the arrangement of the transferred data in internal memory. In the case of CM, the transmit portion of the TCB contains three additional words specifying the ports and duration for the transfer. They too are loaded automatically. With the control registers loaded, the transfer then runs to completion without further intervention by the processor core.

A single TCB describes an elementary transfer, basically moving data from one two-dimensional subarray to another. A composite transfer is specified as a sequence of elementary transfers. Through **auto-chaining** the MeshSP is capable of carrying out an arbitrary sequence of such transfers without further participation on the part of the processor core.

Each channel of a TCB (receive or transmit) contains a word which may be a pointer to the same channel of the next TCB. These words, like other portions of the TCB, are loaded into the DMA control registers. When the current transfer is complete, if these chain pointers are not zero, the control registers are immediately loaded with the contents of the next TCB, initiating the corresponding transfer (see Figure 2-2). The auto-chaining mechanism allows an arbitrary number of elementary CM or SIO transfers to be performed in sequence without interfering with the processor core. This is a key design feature of the MeshSP architecture. The TCB pointer word contains a bit that may be used to cause a hardware interrupt at the completion of the transfer specified by the TCB, which allows coordination of the transfer with arithmetic processing as described in Section 3.2.

2.6 TWO-DIMENSIONAL ARRAYS AND SUBARRAYS

MeshSP algorithms often involve the transfer of multidimensional data arrays between slaves or between the array and host. A common operation involves the transfer of two-dimensional subarrays of two-dimensional arrays, or arrays of complex data, or data downsampled in the x- or y-directions (or both). An example of such a subarray is shown in Figure 2-3. Here, a 3×4 downsampled subarray is defined within an 8×8 parent array. The following parameters are used by the DMA hardware to generate an address sequence.

Addrs	base address of array
Nx	number of points in x-direction
Ny	number of points in y-direction
Dx	x step size
Yinc	y increment

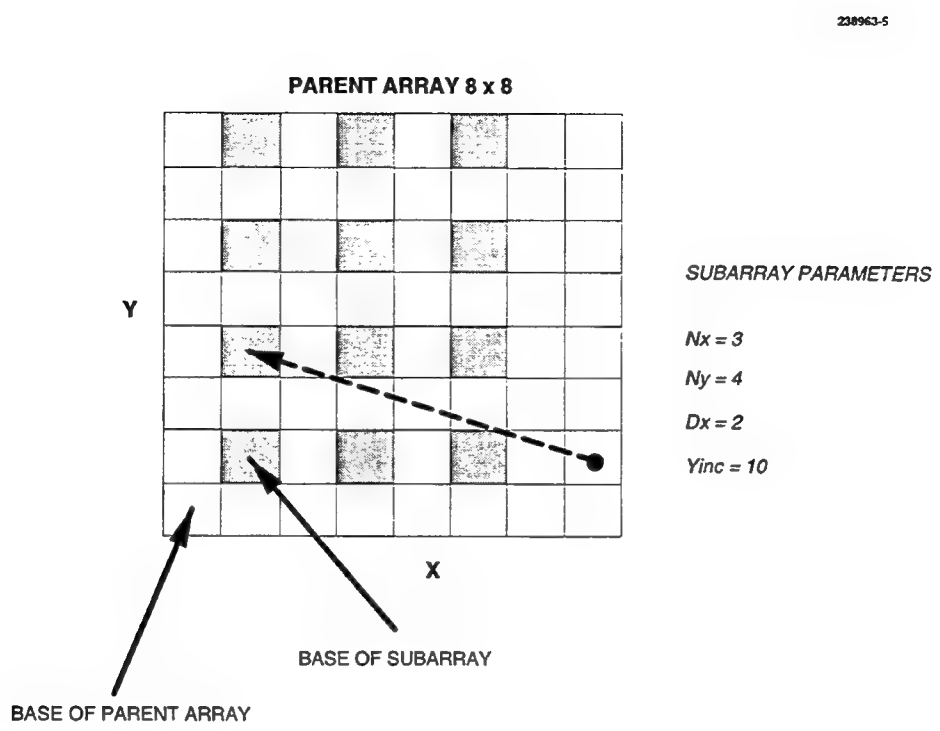

Figure 2-3. A two-dimensional subarray.

Although the first four parameters have simple and direct interpretations, the fifth does not. Yinc is the difference between the address increment when one moves from the end of one row to the beginning of the next, and Dx is the address increment between elements interior to a row. In general

$$Yinc = Dy \times Nx_{parent} - Dx \times Nx \ ,$$

where Dx and Dy are the basic steps in the x- and y-directions, and Nx_{parent} and Nx are the x dimensions of the parent array and subarray. Thus, although Yinc is not as fundamental as Dy, it is easily computed.

The five parameters which define a two-dimensional subarray, together with a chain pointer, comprise the six-word transmit and receive sections for SIO TCBs and receive section for CM TCBs (see Figure 2-2). The CM transmit section contains three additional words that specify the communication I/O ports.

2.7 CM SYSTEM

The MeshSP PEs communicate via a nearest-neighbor, rectangular mesh. The global topology is toroidal (edgeless). For example, each of the eight rows of the MeshSP-1 array forms a ring, as do each of the eight columns. Toroidal connectivity is useful in creating a uniform and seamless computational fabric

and therefore no MeshSP slave is more distant than four x locations and four y locations from any other slave. It is especially useful in certain applications such as the mapping of ground-stabilized data from a moving sensor or performing global operations (such as a global two-dimensional FFT).

Because each slave is provided with six interprocessor (or link) ports, a two-dimensional nearest-neighbor mesh leaves two ports unused. MeshSP-1 uses these extra ports for "column jumping" to provide fault tolerance; they may also be used for more rapid row-directed communications in an intact array. Here, each slave is connected not only to the slaves on the same row and adjacent columns but also to slaves on the same row and two columns over. If either adjacent column fails, it may be bridged by this next-nearest-neighbor link. This link allows the array to degrade gracefully when a slave fails. The array is diminished in size, but no discontinuities remain after the column is bridged. If fault tolerance were not needed, the full set of six links could have been used to form a three-dimensional mesh. The MeshSP concept and software are compatible with any such homogeneous mesh architecture.

2.7.1 CM Link Operation

Each node of the interprocessor communication system consists of the communication register internal to each slave together with its six associated communication links (Figure 2-4). An elementary CM transfer is specified by a direction of transmission, a direction of reception, and a duration. Data move from slave to slave in a series of up to three **legs**. The **duration** of each leg is the length of time during which the identity of the receive port and the transmit port are unchanged. These ports may be reselected between legs. The transfers are fully synchronous in that all slaves load data at the same time, shift at the same time, and unload at the same time. There are six transmit directions that correspond to the six link ports as well as eight receive directions that correspond to the six link ports plus two internal constant registers. A constant register may be used as a substitute for data that otherwise would arrive from one of the physical link directions. This is discussed in the next section.

Communication begins by transferring a word from the internal memory of a slave to its communication register. This word is then passed from slave to slave along a preselected path until it arrives at the destination slave. At that point it is copied from the communication register to the memory of the destination slave. Neither the internal bus nor the memory of intervening slaves are impacted in any way. When the word has been stored at the destination, the DMA hardware initiates transfer of the next word as specified by the TCB.

The memory locations from which data are loaded into (and stored from) the communication register may be completely different for different slaves. The only requirement is for the same number of words to be transferred by each slave.

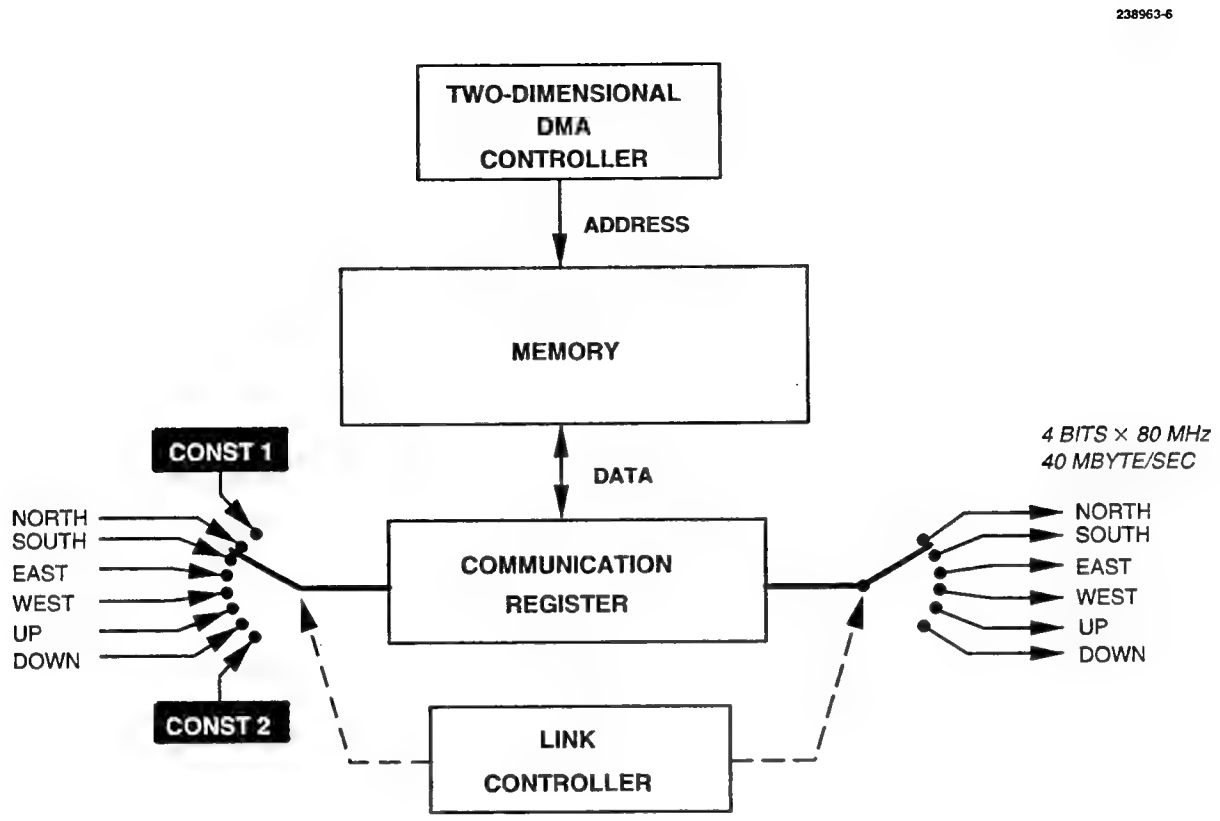

Figure 2-4. CM link.

Communication is fully synchronized between slaves, and the network is kept completely filled. During an ongoing transfer, as a word is shifted from the communication register to the output link, it is replaced by a word arriving from the input link. The physical transfer is nibble-wide (4 bits). Thus, eight CM clocks are required to transfer a single 32-bit word from one slave to the next. Because the CM clock operates at twice the processor clock speed, four CPU clocks are required to transfer a 32-bit word.

For the simplest CM transfers all slaves receive data from one direction and transmit it to the opposite direction, e.g., receive from the west and transmit to the east. In that case if the duration of the leg is N words, each word is shifted N slaves from west to east. In more complex situations the relationship between the leg duration (seen by a fixed slave) and the geometrical displacement (seen by a moving data word) is less direct. The communication illustrated in Figure 2-5, for example, consists of a single leg of duration 2.

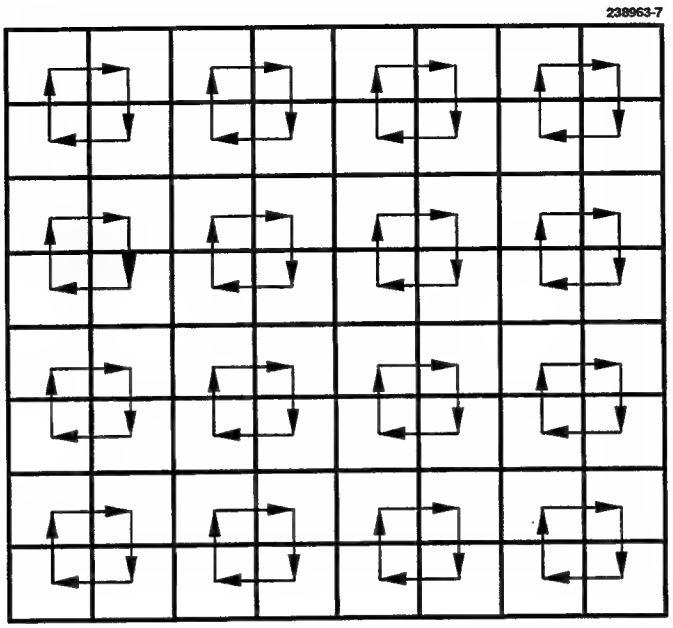

Figure 2-5. A collisionless transfer.

Each slave maintains a single input direction and a single output direction. These directions are not, however, opposite. Thus, the slave in the upper left of each quartet always receives data from below and always sends it to the right. Such a leg of duration 2 interchanges data between opposite corners of the quartet. Moving with the data, one encounters two displacements, e.g., left and down. Staying with a slave, the directions are fixed. The important point is that the leg specification is bound to the slave, not to the moving data. The sole requirement for such transfers for which the directions differ between slaves is that no "collisions" are permissible.

2.7.2 Physical Assignment of Logical Link Ports

There are times when it is useful to alter the basic nearest-neighbor physical connectivity. The MeshSP supports column jumping for fault tolerance. Two of the six link ports may be physically connected to slaves on the same row, but two columns over (the connectivity used in MeshSP-1). If one or more slaves in a given column fail, that column may be effectively removed. Slaves on the two adjacent columns accept data via these auxiliary inputs, bridging the failed column. This reassignment is accomplished by appropriately setting the SHARC link configuration register (LNKC). Once this register is set, all transfers which were originally to come from the failed column will actually come from one slave over. This repair mechanism works for any number of noncontiguous failed columns. For a large array, contiguous failed columns should be more infrequent than isolated or noncontiguous failures. When they do occur, contiguous failures will necessarily result in an unrepairable gap.

2.7.3 Toroidal vs Open Connectivity

Although the MeshSP array is globally toroidal (edgeless), some algorithms require the array to be treated as being embedded in an infinite plane of zeroes. This is called planar or open connectivity. The MeshSP provides for this by allowing certain "edge" slaves to accept data from one of two internal constant registers in place of data arriving from a neighboring slave. The constant may be set to zero or any other number.

Two separate mechanisms have been provided to support this process. Each of the three words of a CM TCB contains a field that specifies whether the received data are to be accepted from the designated logical port or come from one of the two constant registers. This allows the connectivity to be associated with that particular transfer. On the other hand, the link configuration register also contains a field that may be used to map a given logical port to a constant register. In this case, all transfers will be performed in open connectivity as long as the link configuration is so set.

2.7.4 Broadcast Mode and Intraslave Transfers

It is sometimes useful to distribute information from each slave to an entire set of other slaves, e.g., from each slave to all others in the same row or column. Provision has been made to efficiently perform this task in hardware via the **broadcast mode**. Here, the usual load-shift-shift-...-shift-unload sequence is modified by having every shift followed by an unload. For example, a broadcast mode transfer of duration 7 in the x-direction on the 8×8 MeshSP-1 results in a given word being sent to all other slaves in the same row. In this case, the TCB receive section specifies seven times the storage as the TCB transmit section.

The CM system can be used to effect intraslave transfers as well. This is done by selecting the same port for transmit and receive. Such transfers are effectively a single leg of unit duration.

2.8 SERIAL INPUT/OUTPUT SYSTEM

Data are transferred between the array and host via the serial input/output (SIO) system, which is completely independent of the interprocessor CM system. The MeshSP-1 SIO system, comprising eight bit-serial chains of eight slaves apiece, is shown in Figure 2-6. Operating at 40 MHz, it has a transfer rate of 40 Mbytes/sec. Other arrangements are possible by matching the I/O capabilities of the array to system requirements. For example: If a data rate of only 5 Mbytes/sec was sufficient, it would be possible to daisy-chain all 64 PEs. This would reduce the number of connections and simplify the hardware. On the other hand, if a data rate of 320 Mbytes/sec was needed, the serial ports of all PEs could be paralleled. In general, the maximum possible I/O rate is 40 Mbits/sec multiplied by the number of PEs in the array, while the minimum hardware solution provides 40 Mbits/sec total. For many systems intermediate values provide the proper balance of hardware and throughput.

As shown in Figure 2-6, the eight slaves of each column are linked to form a serial chain. The eight chains (one per column) also pass through a custom integrated circuit, the SIO module (SIOM). This chip

forms the physical interface between the slave array and the host processor data bus and is accessible to the host as a set of memory mapped registers.

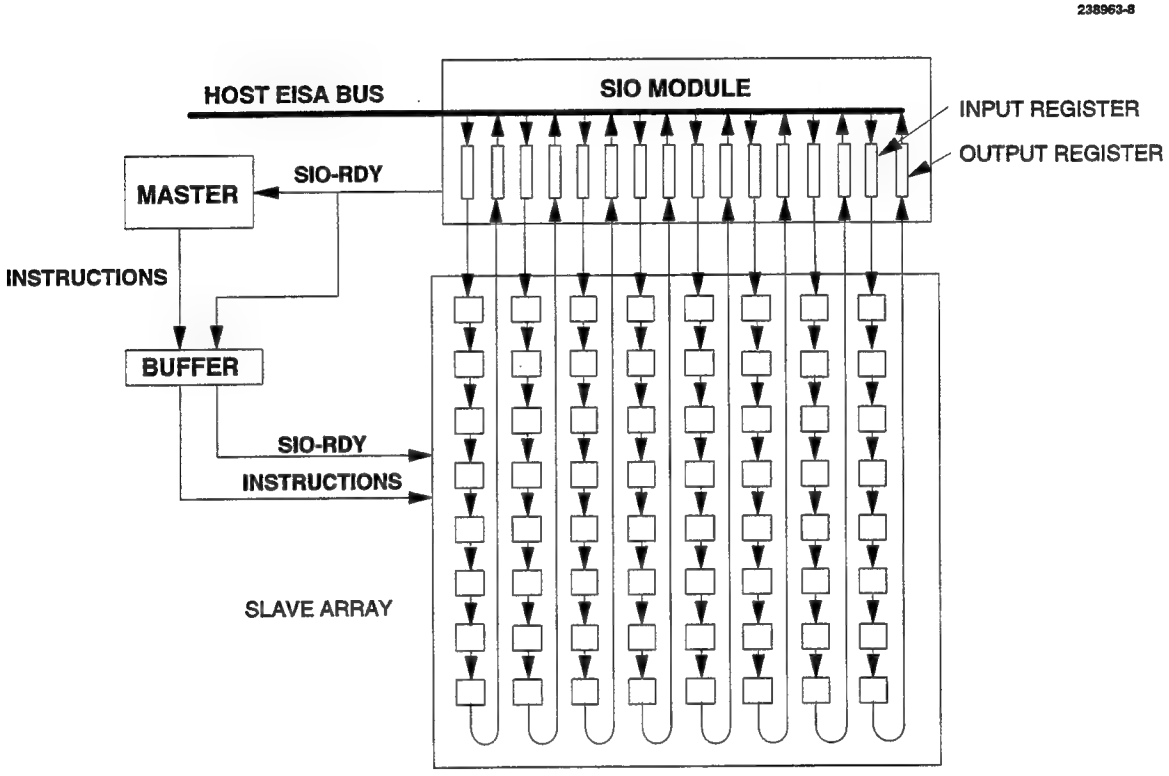

Figure 2-6. The MeshSP-1 SIO System.

2.8.1 SIO System Operation

The basic SIO system operation is to transfer data simultaneously from the MeshSP array to a host output file, as well as to the MeshSP array from an input file on the host.

In preparation for an SIO transfer, the MeshSP program first opens the appropriate files via the master-host interface as described in Section 2.4. In addition, appropriate TCBs are constructed. The program initiates the SIO transfer via a software function which takes as arguments pointers to the files and to the TCBs. The master (and slaves) then load the SIO DMA chain-pointer registers with pointers to the receive and transmit portions of the first TCB. This allows the transfer to begin by loading the slave's SIO register

from internal memory. However, data are not shifted out until a valid SIO-RDY signal is received from the SIOM. The SIOM generates this signal when its input registers are full and its output registers are empty. This step requires cooperation by the host.

The host is informed of the need to read and write these registers via a standard master-host handshake. Through its access to the SIO TCB, the MeshSP master knows how many words must be transferred. This parameter, along with pointers to the host files, is placed in external master memory. The host is then interrupted, forces the master to relinquish the bus, and reads the parameter list. At that point the master is once again allowed access to the bus and becomes active again. The host and the autonomous SIO DMA of the master now begin the body of the SIO transfer.

The host begins by writing the first eight input words to the SIOM, which causes the SIO-RDY signal to become active. This allows each slave serial link to shift out one 32-bit word into the corresponding SIOM output register and to shift in one 32-bit word from the corresponding SIOM input register.

When the eight input words have been shifted out of the SIOM and eight output words have been shifted in, the SIO-RDY signal is deasserted, halting the transfer until the host reads the output words and writes new input words. This process is repeated until the total number of words, previously passed to the host as a parameter, has been transferred.

Access to the SIOM by the host is controlled by a conventional READY signal, which prevents the host from writing SIOM registers that are not empty or reading SIOM registers that are not full.

The operation of the SHARC serial ports requires that every bit clocked in must be accompanied by a bit clocked out. However, one-way transfers are supported by modes in which the SIOM generates its SIO-RDY signal when either the input registers are full or the output registers are empty. In this case the host is not obligated to perform both read and write operations, but one or the other.

2.9 SOME FEATURES OF THE ADSP-21060 SHARC

Analog Devices Inc. provides a detailed and informative user's manual for the SHARC chip, the MeshSP PE, and it is assumed that the reader has access to that document [2]. Although there is no need to duplicate the material of that document, it is worthwhile to emphasize certain of the SHARC's features that are important for MeshSP operation.

2.9.1 Memory Organization and SIMD Operation

Figure 2-7 presents a schematic view of the SHARC organization. As the figure shows, the 4-Mbit internal memory is divided into two 2-Mbit blocks, which presented a potential pitfall for SIMD processing. It is possible in a valid SIMD program for different slaves to access different variables at the same time. This access may be through the use of multivalued pointers or by slave-dependent array indices. The core processor is capable of accessing two variables on a single cycle, providing one variable is passed over the

internal program bus and the other passed over the internal data bus. But each memory block supports only a single access by the core on each cycle.

Thus, if two variables are needed, and if they reside in different blocks, they may be read in a single cycle. If they reside in the same block, two cycles are required. Slave-dependent addressing can therefore result in a loss of synchronization. To prevent this loss from occurring a bit has been provided in the System Configuration Register which may be used to inhibit simultaneous access to the two memory blocks. It is the responsibility of the MeshSP software to set this bit when the potential for desynchronization exists. It is necessary to set this bit only when indirectly accessing arrays, structures, etc., that span the two blocks, information available from the load map.

2.9.2 Data Types

The internal memory of the SHARC is unusual in that it provides for storage of 48-bit instructions (for MIMD processing), 40-bit extended precision floating-point data, 32-bit floating-point and integer data, and 16-bit floating-point data. The 32- and 16-bit data may be freely intermixed within a single block. Forty-eight-bit instructions and 40- and 32-bit data may be stored in the same block, but all instructions and 40-bit data must reside in the lower address locations while 32- and 16-bit data reside in higher address locations. At any time, each block may be configured for 32-, 16-, or 40-bit data access. To avoid excessive toggling of this configuration, computations involving mixed 40- and 32-bit data should separate the data between two blocks.

2.9.3 16-Bit Floating-Point Format

The main objective in developing a monolithic processor element for the Synchronous Processor was to reduce system cost, especially for large systems. Although the SHARC has much more onboard memory than any competitive DSP chip, its 1/2 Mbyte is marginal for applications dealing with large amounts of data. On the other hand, digital signal processing often demands less precision and dynamic range than afforded by the standard 32-bit floating-point format (as the success of 16-bit integer DSP chips attests). Accordingly, the MeshSP processor element was specified to provide instructions for converting floating-point data to and from a 16-bit floating-point format for the purpose of increasing storage capacity.

All SHARC floating-point computations are done internally to 40-bit precision: 8 bits of exponent and 32 bits of mantissa. Results stored in the standard 32-bit floating-point format lose the 8 least significant bits of the mantissa. The 16-bit floating-point storage format uses a 4-bit exponent, effectively allowing a shift of the binary point by 16 locations, for a dynamic range of 96 dB, before losing any precision. The 12-bit mantissa provides over 80 dB SNR against quantization noise. Although not all algorithms can tolerate this lowered precision, it suffices for many signal-processing applications.

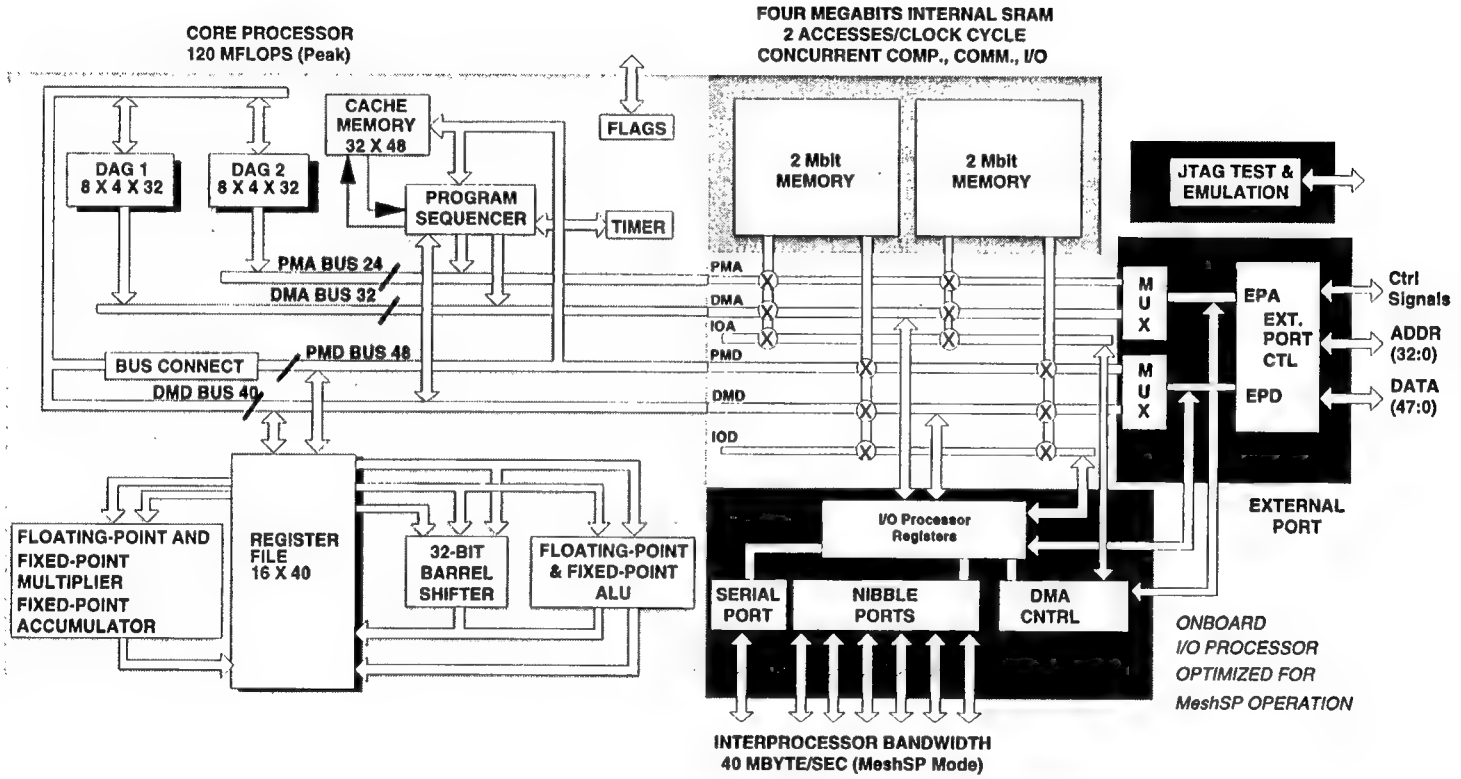

Figure 2-7. Analog Devices' ADSP-21060 (SHARC)

2.9.4 Single-Cycle Operation

As shown in Figure 2-7, the SHARC computational units include a multiplier-accumulator, an ALU, and a 32-bit barrel shifter. These units perform single-cycle operations; there is no computation pipeline. The output of any unit may be the input of any unit on the next cycle. In a multifunction operation the ALU and multiplier perform independent, simultaneous operations. Most arithmetic and logical operations may be paired with concurrent data moves that load or unload the registers to memory. Beyond the usual parallel operations, there is even an instruction that performs the following parallel multiply, add, subtract:

z = x*y, c = a+b, d = a-b;

Here a, b, c, d, x, y, and z are either floating- or fixed-point variables. This triple operation forms the core of the FFT butterfly. The availability of this operation allows full complex FFTs to be performed in $2N\log_2 N$ cycles. It leads to the peak throughput of 120 Mflops and a 1024-point full complex FFT in 0.47 ms.

2.9.5 Instructions that Simplify SIMD Processing

The key to efficient SIMD processing is the avoidance of data-dependent execution times and unnecessary control transfers. The Analog Devices ADSP-21020 has an instruction set very well suited to the SIMD role. This was a major motivation in choosing it as the basis of the MeshSP PE.

Most ADSP-21020 instructions are available in conditional form. Here, the processor checks for one of 32 possible conditions (ALU = 0, ALU < 0, multiplier sign, etc.). If that condition is true, the designated computation and/or data transfer is carried out. If the condition is false the operation is not performed. The entire process takes a single cycle, which qualifies it for SIMD operation. This is a simple and effective way of implementing conditional NULL operations, which are done in a more complex manner in SP-2.

The ADSP-21020 supports single instructions that form the maximum or minimum of pairs of integer or floating-point arguments. In less capable processors this would be done by data-dependent branching, which is inconsistent with SIMD processing. Similarly, there are instructions that clip arguments at specified absolute values.

2.9.6 CM Timing Considerations, Delay, and Skew

To enable the construction of large MeshSP systems it is essential that the CM and SIO systems be relatively insensitive to propagation delay and interprocessor timing skew. This is especially important with regard to CM, as that system operates at a bit rate (80 MHz) which is twice the basic processor clock rate (40 MHz). The SHARC's serial and link ports were designed specifically to meet this requirement. As with the broadcast instruction stream, the serial and link data are accompanied by a clock pulse transmitted along the same path. Moreover, the ports themselves contain special circuitry to allow the I/O processor to maintain synchronism with the core processor even when the communication path introduces significant timing delay

and skew relative to the local clock. The received bits are clocked in by the accompanying clock signal, allowed to settle, and then resynchronized relative to the local clock. The result is robustness to timing errors as great as two cycles (25 ns) in either direction. With good engineering practice, this margin is sufficient to allow construction of systems that extend across multiple boards.

3. MeshSP SOFTWARE

Because parallel programming is often considered difficult, much effort has been expended in developing software tools that attempt to automatically parallelize serial algorithms. This approach has not yet led to any great success. The alternative is to construct algorithms and programs which directly comprehend the parallel organization of the processor architecture. Simple yet efficient programs may be constructed by the explicit and symmetric distribution of processing among equivalent slaves. The responsibility for this decomposition, as well as for explicitly specifying the required interprocessor communication, remains with the algorithm designer. This approach allows use of a standard C compiler with no extensions for parallel operation. Existing serial programs (not including this decomposition) are generally unusable. However, the MeshSP is intended for applications that do justify the design and coding of appropriate parallel algorithms.

A library of functions has been constructed to relieve the programmer of concern with the details of interprocessor communication and I/O. These functions perform such operations at varying levels of abstraction. While programmers are free to control the system at the lowest level, they are also free to use high-level functions which perform quite complex and flexible transfers. As MeshSP coding progresses, other high-level functions which have proven their worth will be added to this library.

The essential process of problem decomposition on the mesh elicits, and often requires, creativity on the part of the algorithm designer. The availability of the MeshSP high-level communication support functions enables rapid assembly of working programs. This frees the designer to concentrate on the issues of parallel decomposition and data distribution. The lack of automatic parallelizing tools is not to be considered a shortcoming, but a benefit. Explicit decomposition is the designer's opportunity to extract maximum performance for the problem of interest. Experienced designers find this the most satisfying part of algorithm development. The novice will find ample assistance in the existing base of MeshSP application code.

3.1 CONSIDERATIONS FOR SIMD PROCESSING

The MeshSP is intended primarily for SIMD operation. This imposes certain constraints on the code. Some constraints are associated with the division of system memory between on-chip (internal) and off-chip (master) memory, while others arise from the need to avoid slave-dependent branching.

3.1.1 Data Storage Classes

These memory constraints are most easily understood in terms of the MeshSP data storage classes. As discussed in Section 2.3, MeshSP variables are assigned to one of three storage classes.

1. **master** located in external memory
2. **single** identical copies located in internal memory
3. **multi** independent copies located in internal memory

Although all MeshSP programs are ordinary C programs, not all legal C programs are compatible with SIMD operation. A set rules governing the use of these data storage classes was constructed, which guarantee that a MeshSP program is compatible with SIMD processing.

The C language makes extensive use of pointer variables, and these must be carefully treated. In addition to the location in which it is stored, a pointer variable has another, independent property: the storage class of the variable to which it points. A MeshSP SIMD program allows any pointer variable to point to one and only one such storage class.

For example, ptr may be a pointer to a multi variable. If ptr is a master variable, it exists in a single copy in the master and points to a common location in the internal memories of the master and all slaves. Because ptr is a pointer to multi, the contents of that common internal memory location may vary from slave to slave. A similar situation obtains if ptr is a single variable, existing in replicated form in all slaves. Finally, ptr might be a multi variable. In that case the various copies of ptr may point to different internal memory locations.

Program code must obey seven rules to ensure correct SIMD functioning with respect to storage class. They are listed below with the help of the following definitions. An expression is multivalued if any component of its construction (variable, index, pointer, etc.) is multivalued. The term **single-valued** refers to either the master or single storage class.

Rule 1. A multivalued expression may not be assigned to a single variable.

This includes either an explicit assignment or a value passed as an argument to a function anticipating a single argument.

Rule 2. A pointer may be assigned the value of another pointer only if both pointers refer to variables of the same storage class.

The storage class of the pointers themselves need not agree, provided rule 2 is obeyed. Thus, a multi pointer to a single variable may be assigned the value of a single-valued pointer to a single-valued variable.

Rule 3. A multivalued pointer may not point to a master variable.

This follows from the observation that only the copy of a pointer to master which resides in the master can access the master variable. To ensure that all copies of this variable maintain the same value, the pointer should either be a master variable or a single variable.

Rule 4. A multivalued pointer pointing to a single-valued variable may be read but not written.

In C terminology, the dereferenced pointer may not be treated as an "l-value". That is, it may be used on the right-hand side of an assignment statement but not the left. It may not be used to return a value from a function via a passed parameter. This is best understood in terms of a specific

example. Suppose all slaves have a copy of the same look-up table in internal memory (a table of single-valued variables). The slaves are free to individually look up values based on their own local data, making use of a multivalued pointer to read these table entries. On the other hand, if the slaves were to write to this table via multi-valued pointers, the table would no longer remain identical in all slaves, i.e., its single-valued character would be compromised.

Rule 5. Data transferred via the master-host interface must reside in external memory.

The host cannot directly access the internal memory of the slave array. Therefore, a write to master internal memory violates the SIMD assumption that a single variable will be reflected in both the master and slaves. A read from master internal memory is physically possible, but we prefer to state the rule in this simpler form.

Rule 6. Automatic variables may not be master variables.

This is because automatic variables are stored on the run-time stack which must reside internal memory.

Rule 7. All members of a union must be of the same stroage class.

This rule prevents a multi member from being used as a single.

Although these rules seem complicated, experience has shown that adhering to them in writing C code is very natural for the programmer who understands the MeshSP architecture.

The MeshSP architecture has implications for the use of structures. The MeshSP tools as well as the ANSI rules concerning structures mandate that structure members be allocated adjacent storage in memory. By contrast, MeshSP internal and external memory occupy disjoint blocks in address space. This precludes a structure from having members stored in both internal and external memory. In other words, all members of a structure must be located in either internal or external memory. A structure may contain both single and multi members. Because of the limited ways that a structure variable may be used, it is necessary only to be cognizant of the classes of its members.

A frequent question regarding SIMD operation concerns the impact of "garbage" data in the master's internal memory. The master, after all, receives no data via the CM or SIO systems. This is of no concern for computation. Because any such data are necessarily multivalued variables, they cannot affect the program flow. Most important, for SIMD operation the PE is operated in a mode where no data dependent run-time exceptions (such as overflow) can affect operation. Thus, by following the rules above integrity of SIMD processing is guaranteed.

The situation is only slightly more complex where CM and SIO are concerned. No difficulty arises when the controlling TCBs are master or single variables. However, the MeshSP supports data access and communication patterns that may be slave dependent. These require TCBs that contain multi variables. If

the master initialized its DMA with invalid parameters, it might overwrite otherwise valid (single) data. It is not difficult to avoid such problems.

For a variety of reasons, each slave may be informed of its location via the SIO system. A pair of multi variables, slave_x and slave_y, contain the coordinates of the slave in the MeshSP array. The master, too, must be aware of its identity. This is supported in hardware by connecting the master SIO input permanently to ground. A convenient way to avoid problems with slave-dependent transfers is to assign the master the slave_x and slave_y coordinates of a valid slave, say (0,0).

3.1.2 SIMD-Compatible Program Flow Rules

The C language includes a number of constructs that control the program flow based on the value of an expression. SIMD compatibility requires the use of single-valued expressions in all cases. The following is a list of C constructs requiring a single-valued expression s in the indicated position.

```
if (s) {}
while (s) {}
for ( ; s ; ) {}
do {} while (s);
switch (s) {}
s ? : ;
(*s)() ;          (pointer to a function)
s ||              (logical OR first argument)
s &&              (logical AND first argument)
```

3.1.3 SIMD-Compatible Surrogates for Data-Dependent Branching

Standard C code often involves branches that depend on the sign of a quantity or relative magnitude of a pair of quantities. For example, one may need to write

```
if (x<0) f=a(x);
else if (x==0) f=b(x);
else f=c(x);
```

Or, as another example:

```
(x<y) ? a(x) : a(y);
```

When the quantities x and y are multivalued, such conditional expressions are incompatible with SIMD processing. However, the MeshSP has several native macros which enable such operations to be performed effectively. These macros make use of the ability of the SHARC processor to perform various comparisons in a SIMD-compatible manner. The following macros accept integer arguments:

```
zero(x)           =  1 if x=0,     0 otherwise.
pos(x)            =  1 if x>=0,    0 otherwise.
```

```
neg(x)          =  1 if x<0,     0 otherwise.
min(x,y)        =  minimum of x and y.
max(x,y)        =  maximum of x and y.
abs(x)          =  absolute value of x.
clip(x,y)       =  x if |x| <= |y|,
                =   |y| if x>0 and |x| > |y|,
                =  -|y| if x<0 and |x| > |y|.
```

A corresponding set of macros, accepting floating point arguments are fzero(x), fpos(x), fneg(x), fmin(x,y), fmax(x,y), fabs(x) and fclip(x,y). The first macros return integer values, while the last four return floats. With the aid of these SIMD-compatible macros, the two examples above may be rewritten.

```
f = a(x)*neg(x) + b(x)*zero(x) + c(x)*neg(-x);
```

and

```
f = a(min(x,y));
```

Note that since all the functions `a()`, `b()` and c() are executed each time, the correspondence with the original conditional example is valid only if the functions do not produce side effects.

3.1.4 Restricted Library Functions

Some ANSI standard functions may be restricted in their use. For example, functions dealing with memory management (malloc) or interrupts (signal) require arguments of class single.

3.1.5 Global Variables and Direction Conventions

In writing MeshSP programs it is often necessary to know a slave's position in the array. This information is contained in global variables defined in the header file msp.h. They are

ARRAY_X_DIM	x dimension of processor array
ARRAY_Y_DIM	y dimension of processor array
slave_x	x position of slave
slave_y	y position of slave
pe_num	ordinal number of slave

If ARRAY_X_DIM = ARRAY_Y_DIM = 8, slave_x and slave_y vary from 0 to 7, while pe_num varies from 0 to 63. In addressing slaves in the processor array, as well as data elements within a slave, we adopt the convention that increasing x or y always correspond to increasing address or pe_num. Furthermore, as address or pe_num increases, the x value varies more rapidly than y. That is, two elements whose y values are equal but differ in x by 1, have addresses or pe_nums that differ by 1. If the x values are equal, but y differs by 1, the addresses or pe_nums differ by the full width of the data or processor array. This is consistent with referring to array elements as A[y][x] (two-dimensional storage) or A[x + y*Nx] (one-dimensional storage) in the C language.

3.2 COMMUNICATION AND I/O SUPPORT SOFTWARE DATA STRUCTURES

Although there are trivial SIMD programs for which the processors execute independently from beginning to end, most programs require the exchange of data among slaves during the course of processing. The functions that support these transfers form the core of the MeshSP software system.

3.2.1 CM and SIO Data Structures

The TCBs used to define MeshSP interprocessor CM and SIO are defined as C language data structures. An SIO TCB is defined as

```
typedef struct sio_tcb
{
    dma t;                      /*   specifies transmit data  */
    dma r;                      /*   specifies receive data   */
}   sio_tcb;
```

It consists of two parts, the first describing the arrangement the data to be transferred from the slave, the second describing the arrangement of data to be transferred to the slave. Each of these descriptions is a structure of the following form.

```
struct dma
{
    void *addrs;                /*   address          */
    int dx;                     /*   x increment      */
    int nx;                     /*   x count          */
    dma *cp;                    /*   next pointer     */
    int yinc;                   /*   y increment      */
    int ny;                     /*   y count          */
}
```

The name of this structure reflects the fact that the SHARC DMA controllers were designed to accept these six parameters directly. The meaning of the parameters is discussed in Section 2.6. A CM TCB is similar to an SIO TCB, but it contains three additional words: the leg descriptors.

```
typedef struct cm_tcb
{
    dma t;          /*          transmit dma structure      */
    int leg1;       /*          1st leg descriptor          */
    int leg2;       /*          2nd leg descriptor          */
    int leg3;       /*          3rd leg descriptor          */
    dma r;          /*          receive dma structure       */
}   cm_tcb;
```

The quantities leg1, leg2, and leg3, denote the one to three possible communication path legs. Each descriptor is a 32-bit word which contains fields for the input direction, output direction, and duration of the leg. The first leg must have a duration greater than 0. The format of each leg is as follows.

bits 0–15	leg duration
bits 16–19	receive link
bits 20–23	transmit link
bits 24–25	specify constant register input
bit 28	broadcast communication mode
bits 29–31	unused

The 3-bit link directions have been arbitrarily defined as follows.

0	1	2	3	4	5
-Y	+X	+Y	-X	+Z	-Z

This convention leads to the following special cases where the transfers are in the x- or y-directions (- to +) or (+ to -).

```
#define PLUS_X                  0x00130000
#define MINUS_X                 0x00310000
#define MINUS_Y                 0x00020000
#define PLUS_Y                  0x00200000
```

A transfer that is purely internal to a slave is accomplished by defining a leg whose input and output links coincide. This may be arbitrarily chosen to be the positive y-direction. Leg1 is specified as

```
#define INTERNAL    1L
```

The 2-bit field, bits 24–25, allows for data to be received from one of two constant registers rather than from another slave. The encoding is

10 use const_1
11 use const_2
00 use receive data

These bits are set in software on the basis of the contents of a structure which holds the connectivity status of the given processor element.

```
struct EDGE
{
    multi int plus_x;
    multi int minus_x;
    multi int plus_y;
```

```
    multi int minus_y;
    multi int plus_z;
    multi int minus_z;
} edge;
```

Each element of edge may be set to 0 to receive data from the corresponding direction or set to 0x01000000 or 0x11000000 to receive the contents of const_1 or const_2, respectively. If bit 28 is set, the communication occurs in a "broadcast mode." That is, instead of a single load, multiple shifts, and a single store, the communication stores the shifted word at the current slave after each shift. This allows efficient distribution of a given word in each slave to all other slaves in the same row or column. In this case the receive data are more numerous than the transmit data by the length of the communication path.

3.3 THE CM CONTROL STRUCTURE

As discussed in Section 2.5, TCBs may be chained together for sequential, autonomous operation. A chain pointer that is not NULL causes the immediate loading of the next TCB. If the interrupt bit of the chain pointer is not set, the core processor remains unaware of this process. If the interrupt bit is set, the core processor is interrupted. Although there is no SHARC hardware requirement linking the interrupt bit to a NULL address field, the MeshSP software enforces that connection through the definition of the chain terminator FINI.

```
#define FINI (dma *)          0x20000L
```

FINI denotes the end of a chain, the finest scale on which CM and SIO are controlled by the core processor. Whether a chain consists of a single or multiple TCBs, the chain pointer of the last TCB is always FINI, which prevents further auto-chaining and interrupts the core processor.

The interrupt service routine performs a number of functions. If other chains are pending, the interrupt service routine will launch the next one. In the CM case, this may involve checking two queues of different priority. It also may arrange to hold the core processor from further computation at a point where the program requires completion of a specified transfer. Finally, the interrupt service routine may call other user-specified computation to be inserted on completion of a particular transfer chain.

This last capability supports a limited form of multitasking that is especially useful when a small amount of computation is directly associated with a communication task. For example, the average (or maximum, etc.) of data distributed across the array could be formed. This is most efficiently done in stages, where intermediate results are computed and then redistributed. Dominated by communication, this process may be effectively overlaid with another purely arithmetic task. From the programmer's point of view, combining the communication and computation portions of the global average allows them to be initiated with a single function call and run to completion with maximum concurrency.

The CM system control structure which supports these capabilities is shown in Figure 3-1. It consists of a pair of ring buffers (one for each priority), an integer, CM_status, and an interrupt vector table,

INT_VECT_TABLE, for use by the multitasking system described above. The status word indicates whether a transfer is running and its priority.

CM_status = 0	No CM transfer in progress
CM_status = 1	Low-priority transfer in progress
CM_status = 2	High-priority transfer in progress

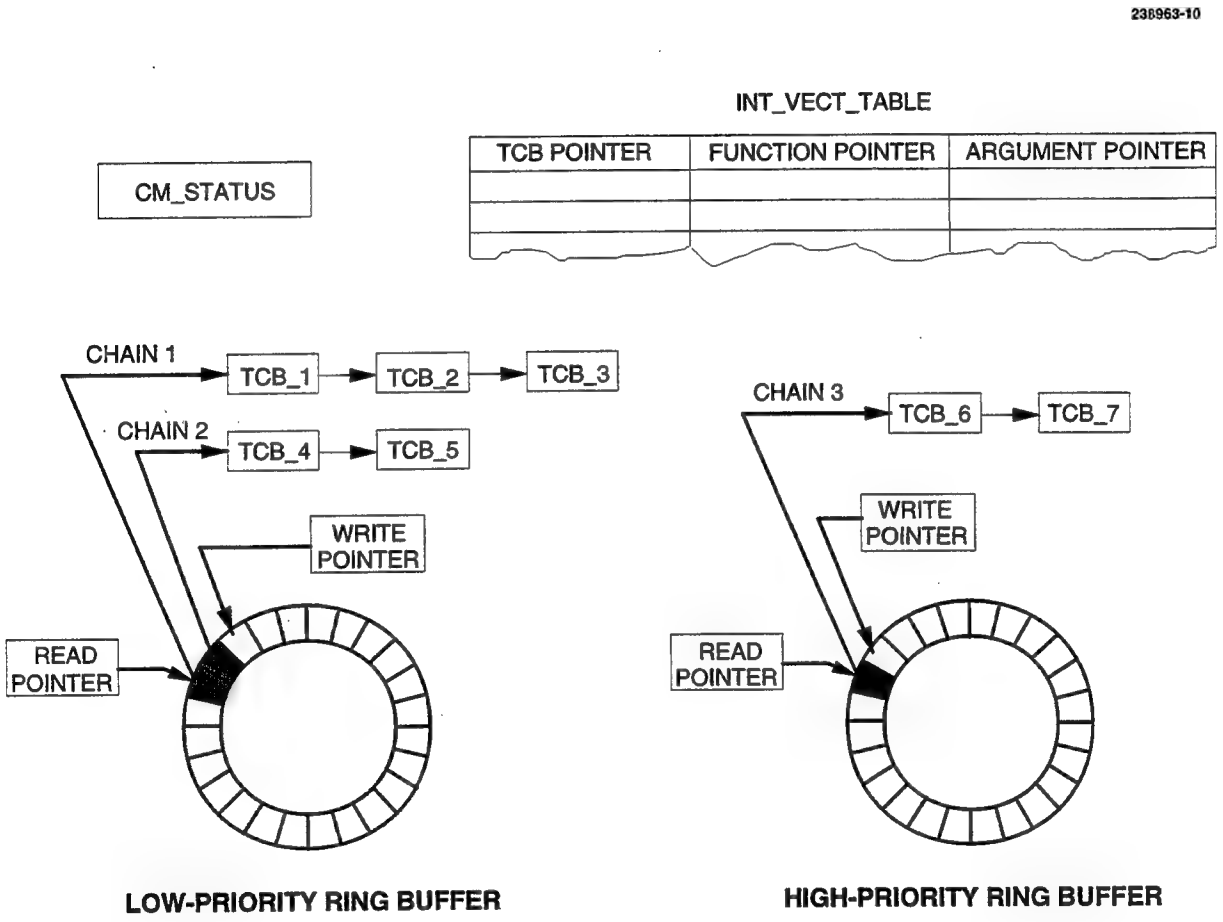

Figure 3-1. CM control system.

The ring buffers maintain a pair of queues of pointers to TCB chains. A pointer to a TCB is maintained on the appropriate queue until the transfer is complete. While a transfer is in progress, the

appropriate read pointer points to the TCB at the head of the chain. Pointers to other chains, pending execution, are further back on the queue.

When a chain is completed, the interrupt service routine first checks to see whether there is an entry in INT_VECT_TABLE with a TCB pointer that corresponds to the transfer just completed. If so, the corresponding function is called (with a single argument) the pointer, which is the third entry in the table. The fact that the interrupt function takes a single pointer argument is really no restriction because that pointer can point to an arbitrarily complex structure.

When the called function returns (or immediately if the table contains no entry with a TCB pointer matching the completed transfer), the current TCB pointer is deleted from the circular buffer by advancing the read pointer. If both queues are empty, the routine simply returns. Otherwise, another transfer is initiated. If there is anything in the high-priority queue, the next high-priority transfer is initiated. If the high-priority queue is empty, the next low-priority transfer is initiated. The status word is then set to indicate whether a transfer is in progress, and if so, its priority.

The CM priority queues and the CM interrupt vector table can alter the temporal execution sequence of routines from the order in which they appear in the source program. This is useful for maximizing concurrency between communication and computation. Concurrency can be managed without these tools by carefully ordering the invocation of functions. Such manual ordering is difficult for two reasons. First, a knowledge of the timing of each arithmetic function or CM transfer chain is required to determine the ordering. Subsequent modifications to the algorithm force a code rearrangement to accommodate new timings. Second, the desire for code modularity implies that some operations requiring both communication and computation should be packaged as units without knowledge of the context in which they will be called.

Judicious use of the priority queues and interrupt vectors helps the programmer achieve a high degree of concurrency while maintaining the algorithm's logical structure in the source code. High-priority CM is useful in situations where particular transfer must be completed before subsequent arithmetic proceeds. Conversely, the CM interrupt vector table can be used to force execution of a computational task which must follow one communication chain and precede a subsequent chain.

The SIO control system is essentially identical to that of CM, with the simplification of a single priority and no interrupt vector table. The SIO status word is either 1 or 0, depending on whether an SIO transfer is in progress.

3.4 CM AND SIO SOFTWARE FUNCTIONS

Three levels of CM and SIO software can be distinguished.

1. Low-level functions directly for accessing the hardware

2. Midlevel functions for creating and modifying data structures

3. High-level function for convenient access to complex communication and I/O patterns

These functions are sufficiently important in defining the nature of the MeshSP that they will be described in full.

3.4.1 Low-Level CM and SIO Functions

```
int CM(
    int priority,              /*    LOW or HIGH              */
    cm_tcb *chain_ptr )        /*    pointer to CM chain      */
```

`CM()` is used to initiate transfers between slaves via the communication system. It returns the value 0 if the transfer starts immediately, 1 if the transfer is pending, and -1 if the ring buffer for that priority is full.

`CM()` first disables the CM hardware interrupt to prevent any new chains from being initiated by the interrupt service routine. If the ring buffer at the requested priority is full, the interrupt is enabled, and the function returns a value -1, indicating an error condition. If the ring buffer is not full, it is written with the `chain_ptr`, and the write pointer advanced. If no transfer is currently in progress, the requested transfer is started immediately by writing `chain_ptr` to the transmit DMA chain-pointer register, and the `chain_ptr` + 9 to the receive DMA chain-pointer register. The offset of 9 corresponds to the specification of a `cm_tcb` as a nine-word transmit portion and a six-word receive portion.

```
int WaitCM (cm_tcb *chain)     /*    pointer to head of chain    */
```

`WaitCM()` provides synchronization of processor and CM activities. This may be required either to ensure that needed data are available for a computation or to verify that an output buffer is free to be reused. `WaitCM()` takes a single argument, a pointer to the TCB at the head of the CM chain. When the processor arrives at `WaitCM()`, it continuously checks to see if the referenced TCB is presently active on either ring buffer. If so, it waits until the transfer is completed (and the entry disappears). If the referenced TCB is not present on a ring buffer, the function returns immediately. `WaitCM()` returns 1 if the designated transfer was in progress when it was called, and 0 if it was not.

Another use of this function concerns temporary TCBs, either created as automatic variables on the system stack via a function call or by explicit dynamic memory allocation. It is important to note that the TCB must survive intact not merely until after the `CM(TCB)` function call but until after the transfer actually completes. This is ensured by means of the `WaitCM()` function. A function that creates a TCB as an automatic variable should not return without first executing a `WaitCM(TCB)`, and if the TCB is created via `malloc()`, it should not be freed before a corresponding `WaitCM(TCB)`.

```
int         SIO (sio_tcb *chain_pointer);
int         WaitSIO (sio_tcb *chain)
```

These functions are similar to their CM counterparts, with the simplification of a single priority.

```
int     SetVect(
        cm_tcb *TCB,                    /*   pointer to tcb          */
        void (* func) (void *),         /*   pointer to function     */
        void *arg )                     /*   pointer to argument     */
int     FreeVect(
        cm_tcb *TCB )                   /*   pointer to tcb          */
```

As previously described, the MeshSP software supports insertion of computational code between communication chains. The function `SetVect()` is used to add a function to the interrupt vector table, and `FreeVect()` to remove it. A function to be added to this table must take a single argument, a pointer to void. If a more complex set of arguments is required, these arguments may be first gathered into a single structure, and the address of the structure is passed as the argument. The inserted code should not initiate or wait for CM.

```
    void InputOutput(
        void  *infile,    /*      host input file                 */
        void  *outfile,   /*      host output file                */
        int  size,        /*      number of bytes per slave       */
        void *indata,     /*      pointer to slave input data     */
        void *outdata )   /*      pointer to slave output data    */
```

`InputOutput()` is a simplified I/O call. It assumes that the data to be simultaneously input and output are in contiguous blocks of "size" bytes. This function prepares the necessary TCB and calls SIO() to perform the desired transfer.

```
    void Input(
        void *file,       /*      host input file                 */
        int  size,        /*      number of bytes per slave       */
        void *data )      /*      pointer to slave input data     */

    void Output(
        void *file,       /*      host output file                */
        int  size,        /*      number of bytes per slave       */
        void *data )      /*      pointer to slave output data    */
```

`Input()` and `Output()` are similar to `InputOutput()` but transfer data in one direction only.

3.4.2 Functions for Preparing TCBs and Chains

```
void MkDma(
    multi dma *DMA,  /*  pointer to dma descriptor           */
    void *base,      /*  base address of parent array       */
    int Nx,          /*  parent array dimension             */
    multi int x,     /*  x pos.of subarray relative to base */
    int dx,          /*  x step                             */
    int nx,          /*  number of elements in x-direction  */
    multi int y,     /*  y pos.of subarray relative to base */
    int dy,          /*  y step                             */
    int ny )         /*  number of elements in y-direction  */
```

`MkDma()` is an intermediate-level function that fills in the elements of a dma structure on the basis of a higher-level description of the data arrangement. It accepts a description of the parent array in terms of its base address and nondownsampled width in the x-dimension. The subarray is specified in terms of its (x,y) location relative to the parent base, the degree of down-sampling in the x- and y-direction, and the number of elements in each direction. From these it determines the necessary dma parameters. The array is traversed first in x then in y. That is, the elements in each row (constant y) are addressed before the elements in an adjacent row. If the increments dx and dy are negative, the rows or columns are traversed backward (in the direction of decreasing address).

```
void MkDmaY(
    multi dma *DMA,  /*  pointer to dma descriptor           */
    void *base,      /*  base address of parent array       */
    int Nx,          /*  parent array dimension             */
    multi int x,     /*  x pos.of subarray relative to base */
    int dx,          /*  x step                             */
    int nx,          /*  number of elements in x-direction  */
    multi int y,     /*  y pos.of subarray relative to base */
    int dy,          /*  y step                             */
    int ny )         /*  number of elements in y-direction  */
```

`MkDmaY()` is similar to the function `MkDma()`. It differs only in that columns are traversed before rows. Rows and columns may be interchanged by using `MkDma()` for the transmit dma, and `MkDmaY()` for the receive dma (or vice versa).

```
void MkDmaC(
    multi dma *DMA,  /*  pointer to dma descriptor            */
    void *base,      /*  base address of parent array         */
    int Nx,          /*  parent array dimension               */
    multi int x,     /*  x pos.of subarray relative to base   */
    int dx,          /*  x step                               */
    int nx,          /*  number of elements in x-direction    */
    multi int y,     /*  y pos.of subarray relative to base   */
    int dy,          /*  y step                               */
    int ny )         /*  number of elements in y-direction    */
```

MkDmaC() is used to construct a dma structure to address complex data. In this case, the complete two-dimensional flexibility for scalar variables is not available. MkDmaC() may be used only for subarrays that are either not downsampled in x or else are not downsampled in y and are of full width in x (parent width equals subarray width).

```
void Connectivity( int con )   /* constant defining connectivity */
```

Connectivity() is used to produce either open or toroidal connectivity. Connectivity(TOROIDAL) ensures that subsequent CM TCBs are constructed so that data fully wrap around the array, i.e., data leaving the top edge arrive at the bottom and data leaving the left edge arrive at the right. Connectivity(OPEN), on the other hand, ensures that data arriving at the edges of the array are zeroes.

```
void XYPath(
    cm_tcb *TCB,     /*  pointer to CM TCB          */
    int xleg,        /*  (signed) x distance        */
    int yleg )       /*  (signed) y distance        */
```

XYPath() sets the leg descriptors in a cm_tcb for simple rectilinear transfers in the x- and y-directions.

```
void LinkCM(
    cm_tcb *TCB1,    /*  first TCB        */
    cm_tcb *TCB2 )   /*  second TCB       */
void LinkSIO(
    sio_tcb *TCB1,   /*  first TCB        */
    sio_tcb *TCB2 )  /*  second TCB       */
```

LinkCM() and LinkSIO() set the dma pointer of the first TCB to point to the second. This permits the construction of extended communication and I/O chains which are transferred in their entirety without intervention by the core processor.

```
void EndCM (cm_tcb *TCB)
void EndSIO (sio_tcb *TCB)
```

It is essential that any chain, even if consisting of a single TCB, be terminated properly. That is, the final dma chain pointers must be FINI. The functions `EndCM()` and `EndSIO()` simply put FINI into the chain pointers of the referenced TCB.

3.4.3 Data Extraction and Insertion Functions

The MeshSP software provides a convenient and uniform high-level mechanism for exchanging data between the host and the MeshSP array or master. The data extraction functions allow results to be captured for later display and analysis, while the data insertion functions provide a means for data to be injected into the processing stream at various points. The functions `Extract`, `ExtractM`, `Insert`, and `InsertM`, which provide these services, may be distributed throughout the code, but activated on any particular run only as desired. They are controlled by means of a system of file pointers.

The MeshSP library declares two arrays of file pointers, `InFile[]` and `OutFile[]`, which are stored in master memory. The dimensions of these arrays are declared in cm_sio.h as MAX_IN_ID and MAX_OUT_ID, respectively (currently set to 256). These arrays must be declared as externs in the application code.

```
extern FILE *OutFile[];  /*if extraction is to be performed*/
extern FILE *InFile[];   /*if insertion is to be performed*/
```

Each call to one of the extraction or insertion functions specifies an ID parameter. The operation is performed only if the corresponding file pointer is not NULL, i.e., set to a legitimate host file. Thus, initialization code may enable or disable data extraction and insertion, as well as select the file which is to receive or transmit data. To enable data extraction associated with a specific ID, one assigns `OutFile[ID]` to a valid pointer to a previously opened file. By assigning NULL to `OutFile[ID]` this particular data extraction is inhibited.

Each data message consists of a fixed size header containing five fields and a variable length data field. The header is defined by the following structure.

```
typedef struct edm_header
{
    int ID;           /* message identifier                  */
    int size;         /* total size in bytes (all slaves)    */
    char type[20];    /* type descriptor                     */
    int Ny;           /* number of data rows (per PE)        */
    char legend[96];  /* legend string                       */
}edm_header;
```

The second element of the header, size, represents the total quantity of data to be transferred to the file, i.e., the sum of the data items (in bytes) present in all slaves. The third element, type, is a string which must correspond to a currently defined C data type. This may be a basic type such as "float", "int", etc.,

or it may be a data type defined in a typedef statement, e.g., "complex" or "cm_tcb", or some other user-defined type. The fourth element, Ny, is used to support two-dimensional arrays, which is a very common MeshSP data type.

The functions that extract and insert data are

```
void Extract(
    int ID,             /*  extraction message identified     */
    char *type,         /*  type descriptor                   */
    int Nx              /*  x dimension                       */
    int Ny              /*  y dimension                       */
    char *legend,       /*  to legend                         */
    multi void *data )  /*  slave array data to be extracted  */

void ExtractM(
    int ID,             /*  extraction message identifier     */
    char *type,         /*  type descriptor                   */
    int Nx              /*  x dimension                       */
    int Ny              /*  y dimension                       */
    char *legend,       /*  pointer to legend                 */
    master void *data )/*   master data to be extracted       */

int Insert(
    int ID,             /*  extraction message identifier     */
    char *type,         /*  type descriptor                   */
    int Ny,             /*  y dimension                       */
    multi void *data )  /*  slave array data to be inserted   */

int InsertM(
    int ID,             /*  extraction message identifier     */
    char *type,         /*  type descriptor                   */
    int Ny,             /*  y dimension                       */
    multi void *data )  /*  master data to be inserted        */
```

The functions Extract and Insert are used to transfer data between the slave array and disk files, while ExtractM and InsertM perform the corresponding functions between master memory and disk files.

The arguments of these functions are related to, but somewhat more convenient, than the quantities appearing directly in the header file. They are actually modified by the C preprocessor by the following redefinitions in the header file **cm_sio.h.**

```
#define Insert(ID, type, Ny, data)    \
        insert(ID, #type, Ny, data)
```

```
#define Extract(ID, type, Nx, Ny, legend, data)    \
        extract(ID, (Nx)*(Ny)*sizeof(type), #type, Ny, legend, data)
#define InsertM(ID, type, Ny, data)    \
        insertM(ID, #type, Ny, data)

#define ExtractM(ID, type, Nx, Ny, legend, data)    \
        extractM(ID, (Nx)*(Ny)*sizeof(type), #type, Ny, legend, data)
```

where `extract()`, `insert()`, `extractM()`, and `insertM()` are the functions that appear in the library, as opposed to `Extract()`, `Insert()`, `ExtractM()`, and `InsertM()`, which are the function calls that appear in the application code.

Note that the second argument of `Extract()` and `ExtractM()`, type, appears in the function call without surrounding quotes. This allows the preprocessor to use it as an argument for the compile time function `sizeof()`, and in conjunction with Nx and Ny, determine the number of data bytes per slave. The function `extract()` then uses the global variable ARRAY_SIZE (total number of slaves) to determine the total size of the recorded data. The header also requires type to be converted into a character string for insertion into the header. This accomplished by the preprocessor symbol #.

The fifth argument of `Extract()` and `ExtractM()`, legend, is an optional descriptive label. It is passed on unmodified to the extraction function.

3.4.4 High-Level CM Functions

A number of communication patterns are commonly used in MeshSP applications and have therefore been packaged as general-purpose library functions.

```
void Augment(
     multi float *r,  /*  pointer to destination data       */
     multi float *t,  /*  pointer to source data            */
     int nx,          /*  x dim. of unaugmented data       */
     int ny,          /*  y dim. of unaugmented data       */
     int dx,          /*  columns to be added each side    */
     int dy )         /*  rows to be added top and bottom  */
```

`Augment()` is a high-level function which expands an original (unaugmented) array, which brings in data from neighboring slaves. It is used primarily to support segmented convolution, a technique for filtering data that has been divided among the slaves. The process is described in more detail in Section 5-2.

`Augment()` is written in general form and will reach out to as many slaves as necessary to obtain the required data. It augments first in the x-direction, then the y-direction, as shown in Figure 3-2. The source and destination arrays are nonoverlapping and of differing dimension: nx×ny for the source and (nx+2dx)×(ny+2dy) for the destination.

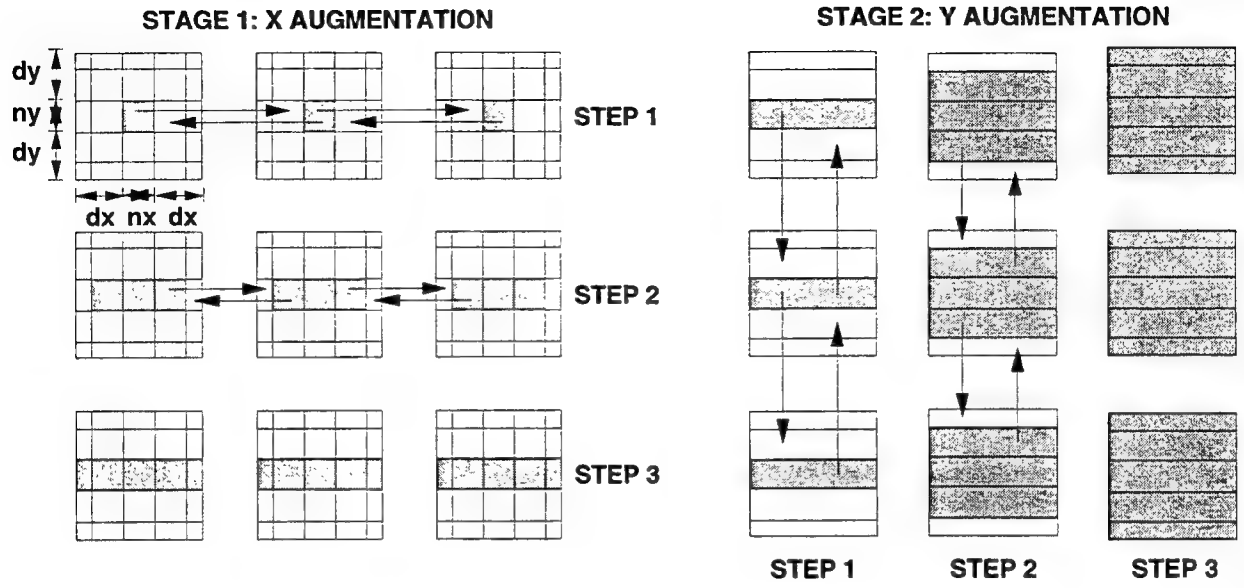

Figure 3-2. Augmentation procedure.

```
void Excise(
        int mode,          /*  mode = SET or ADD                    */
        multi float *r,    /*  excised array                        */
        multi float *t,    /*  augmented array                      */
        int nx,            /*  x dimension of excised array         */
        int ny,            /*  y dimension of excised array         */
        int dx,            /*  columns (each side) to be trimmed    */
        int dy )           /*  rows(top and bottom) to be trimmed   */
```

`Excise()` is essentially the inverse of `Augment()`. It removes the central portion of a rectangular array and either places it in another smaller array or else adds the contents of the central portion to the smaller array. In the first case the function is called with `mode=SET`, in the second case `mode=ADD`. The arrays need not be different, but in most cases will be because the excised array will be dimensioned nx×ny, while the augmented array will be dimensioned (nx + 2dx)×(ny + 2dy).

```
void GlobalAugment(
     multi float *r,  /*  pointer to destination data          */
     multi float *t,  /*  pointer to source data               */
     int nx,          /*  x dim. of unaugmented data          */
     int ny )         /*  y dim. of unaugmented data          */
```

This function is a variant of Augment(), which causes a data array r to be created by importation of data from all other slaves in the array. This new array is the original data array t, as distributed across the entire processor array. It is identical in all processors. If the array t has the dimension nx×ny, the array r will have the dimension (ARRAY_X_DIM×nx)×(ARRAY_Y_DIM×ny).

```
void Shift(
     multi float *r,  /*  pointer to destination data          */
     multi float *t,  /*  pointer to source data               */
     int nx,          /*  x dimension of array                */
     int ny,          /*  y dimension of array                */
     int dx,          /*  x shift (cell)                      */
     int dy )         /*  y shift (cells)                     */
```

Shift() moves an array, distributed across all slaves, by dx in the x-direction and dy in the y-direction (Figure 3-3). The function imports data from as far as necessary to perform the shift. The source and destination arrays must be nonoverlapped; otherwise, data will be overwritten.

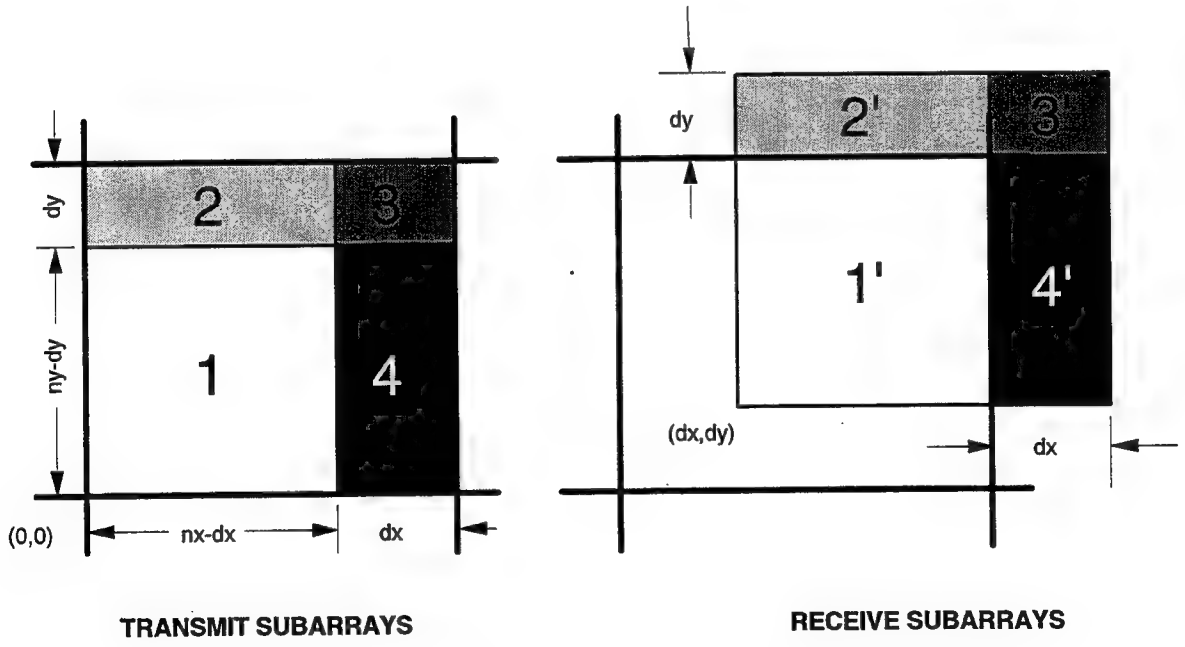

Figure 3-3. Shifting algorithm.

```
void Permute_X(
   cm_tcb *tcb,    /* ptr to first of a chain of ARRAY_Y_DIM tcbs    */
   int sign,       /* determines the direction of permutation       */
   dma dma1,       /* describes first subarray                      */
   int step1,      /* address offset between subarrays              */
   dma dma2,       /* describes second subarray                     */
   int step2 )     /* address offset between subarrays              */

void Permute_Y(
   cm_tcb *tcb,    /* ptr to first of a chain of ARRAY_Y_DIM tcbs    */
   int sign,       /* determines the direction of permutation       */
   dma dma1,       /* describes first subarray                      */
   int step1,      /* address offset between subarrays              */
   dma dma2,       /* describes second subarray                     */
   int step2 )     /* address offset between subarrays              */
```

Permute_X() and Permute_Y() have been found to be valuable in many apparently unrelated contexts, e.g., two-dimensional Fourier transformation, global addition of arrays, and supervised learning in multi-layered neural networks. These functions perform generalized permutations in the row and column directions, respectively. The nature of these permutations may be understood most easily by first considering a simple special case.

Suppose each slave contains a one-dimensional numerical array whose dimension equals the number of columns in the processor array (ARRAY_X_DIM). Each element may then be thought of as being identified by three indices, two indices designating the slave row and column, the third designating the element of the data array. Let A(i)(j)[k] represent the k^{th} element of the data array A in the i^{th} slave row and j^{th} slave column. The communication pattern that performs the transform A(i)(j)[k] -> A(k)(j)[i] is called a row permutation, while A(i)(j)[k] -> A(i)(k)[j] is called a column permutation.

As actually implemented, considerably greater generality is provided. The transmitted data need not be a set of ARRAY_X_DIM simple elements, but it may instead be a set of ARRAY_X_DIM subarrays, the shape of which is defined by the contents of dma2 if sign=1, or dma1 if sign=-1. Correspondingly, the received data may be differently arranged, their shape being defined by dma2 if sign=1 or dma1 if sign=-1. By linking the roles of dma2 and dma1 to the sign parameter, the permutation operations are reversed by simply negating sign.

This flexibility is useful in a variety of applications. In particular, the function GlobalFFT() uses it to rearrange complex rows or columns so that each row or each column is packed into a single slave in preparation for one-dimensional Fourier transformation.

```
void Spread_X( multi void *r_data,   /* pointer to receive data              */
      multi void *t_data,            /* pointer to transmit data             */
      int n )                        /* number of words to be transmitted     */

void Spread_Y( multi void *r_data,   /* pointer to receive data             */
               multi void *t_data,   /* pointer to transmit data            */
               int n )               /* number of words to be transmitted   */
```

The functions Spread_X() and Spread_Y() communicate the contents of a block of data within a slave to all slaves in the same processor row or column, respectively. The number of transmitted words is n, and the number of received words is then either n*ARRAY_X_DIM or n*ARRAY_Y_DIM. The communication uses toroidal connectivity.

The received data are arranged in cyclic fashion, with data from the reference slave appearing first, followed by data from the slave at the next higher column (or row), etc. The last block of data is that which was received from the slave at the next lower column (or row).

As MeshSP coding progresses it is anticipated that additional high-level communication functions will be established in the MeshSP software library.

3.5 STRING AND CHARACTER DATA FOR MASTER-HOST INTERFACE FUNCTIONS

The master-host interface functions provide the connection between data stored in the MeshSP master and I/O streams maintained in the host. These streams include disk files, keyboard input, and CRT output. The basic mechanism of the master-host interface has been described in Section 2.2. Through it the MeshSP is provided with essentially all standard C I/O functions via host resources. One area that requires special attention is the difference between the treatment of characters and strings in the 32-bit SHARC and the byte-oriented host PC.

The basic character type of the PC host is the 8-bit byte. Strings are sequences of such characters, and each 32-bit double word can contain four characters. This convention is supported in hardware by byte addressability. The 32-bit SHARC chip does not provide byte addressability, and the basic character size is 32 bits wide. SHARC strings are sequences of 32-bit words.

All strings stored in MeshSP memory, master or slave, adhere to the SHARC convention and maintain each character in its own 32-bit word. This allows any string to be accessed by the ANSI C string functions provided with the SHARC C compiler.

On the other hand, all strings maintained in host memory adhere to the standard 8-bit byte convention, which allows them to be accessed by host hardware and software without modification.

Generally, the origin and destination of character strings is unambiguous. Consider, for example, the function fgets(s, n, file), which reads at most the next n-1 characters from the stream file into the array s. The function understands that the 8-bit characters in file must be converted to 32-bit characters before

storage in s (in MeshSP memory). Similarly, the format string appearing as the first argument in a printf() statement is understood to reside in MeshSP memory as 32-bit characters.

The only possible ambiguity concerns the functions fread() and fwrite(). These functions are passed pointers to MeshSP memory buffers without consideration of their contents. Accordingly, they make no conversion. If the buffer contains a string of 32-bit characters, it will be stored as 32-bit characters in the host file. If conversion is desired, a function that explicitly recognizes character strings [such as fputs()] must be used.

4. FUNCTIONAL SIMULATOR

The functional simulator is a program that executes MeshSP application code. At the present time the simulator runs on an OS/2 platform, although it may be ported to other platforms in the future. In this section the simulator's purpose and internal design will be discussed.

4.1 PURPOSE OF THE SIMULATOR

From the beginning, the convenience, productivity, and early availability of the MeshSP software development environment have been a concern. In particular, there was a requirement to begin coding of support libraries and application code about a year and a half before the MeshSP hardware was to be available. This was facilitated by the MeshSP simulator. Although slower than the MeshSP hardware by more than three orders of magnitude, the simulator and associated commercial development tools provide an exceptionally productive programming environment.

The MeshSP simulator provides functional simulation for the MeshSP architecture. Functional simulation means that it reproduces the functions of the MeshSP hardware with enough fidelity to permit algorithm development, coding, and debugging to be done independently of the hardware. This work can proceed on relatively inexpensive workstations.

The MeshSP simulator

1. Accepts the same C language application source code as the MeshSP hardware
2. Simulates the results of SIMD computation for the slave array
3. Reproduces the effect of interslave communication
4. Reproduces the effect of I/O with the host computer

These differences between the simulator and MeshSP hardware should be understood.

1. The simulation is carried out on a single processor; it is not literally parallel.
2. No details of the SHARC hardware (registers, multipliers, etc.) are simulated.
3. The simulation is not bit for bit; floating-point computations may differ both in round-off and in precision (word size).
4. The intermediate states of interslave transfers and I/O are not faithfully reproduced. The simulation does not reproduce the mechanics of data movement from one SHARC chip to the next; only the final effect of the transfer is reproduced.
5. The simulation is very much slower than the real-time hardware.
6. The simulator provides no timing information at all.

The simulator provides some services not available with the MeshSP hardware alone.

1. The simulator warns about certain coding problems, such as illegal TCBs and inconsistent interslave link connections

2. The simulator enables the power of modern commercial debugging software to be applied to the application on a slave-by-slave basis. This provides more visibility into the MeshSP code and data than is possible with the MeshSP hardware.

4.2 DESIGN OF THE OS/2 SIMULATOR

Our initial MeshSP simulator effort is based on the OS/2 operating system platform. OS/2 was chosen because of its robust multitasking capabilities on inexpensive platforms. This section specifically references the OS/2 version.

4.2.1 Operating System Concepts

The IBM OS/2 operating system is a 32-bit and multitasking system that is capable of running many tasks simultaneously, each with its own memory and context. The operation of the simulator is best described with the help of some operating systems terminology.

A **process** is a task with its own protected area of memory that executes independently and asynchronously with respect to other processes.

A **thread** is a task that executes asynchronously with respect to other threads. It shares resources with other threads in the same process, such as global variables. Every process consists of one or more threads.

A **pipe** is an area in "shared" memory accessible to multiple processes, through which the processes pass messages or data to each other. Pipes are self-synchronizing in the sense that a read from a pipe will hold execution of that process until another process writes the expected data to the pipe.

A **semaphore** is a word of "shared" memory accessible to multiple processes through which the processes may pass a signal. Execution may wait for a semaphore signal, or a semaphore may simply be queried by a process with access to it.

4.2.2 Design Considerations

The process is a natural analogy to the MeshSP slave in the sense that it has its own area of memory that is protected from access by other processes. Furthermore, it is possible for multiple processes to execute from the same copy of the program, as MeshSP slaves execute the same (broadcast) code. This means there is no need for the memory overhead of program replication in the simulator. Unlike MeshSP PEs, processes run asynchronously. Process execution is timesliced into the processor by the operating system. All such scheduling issues can be left to the operating system as long as the simulator can impose any synchronization required for proper operation of interslave transactions. Interslave communication and SIO are naturally

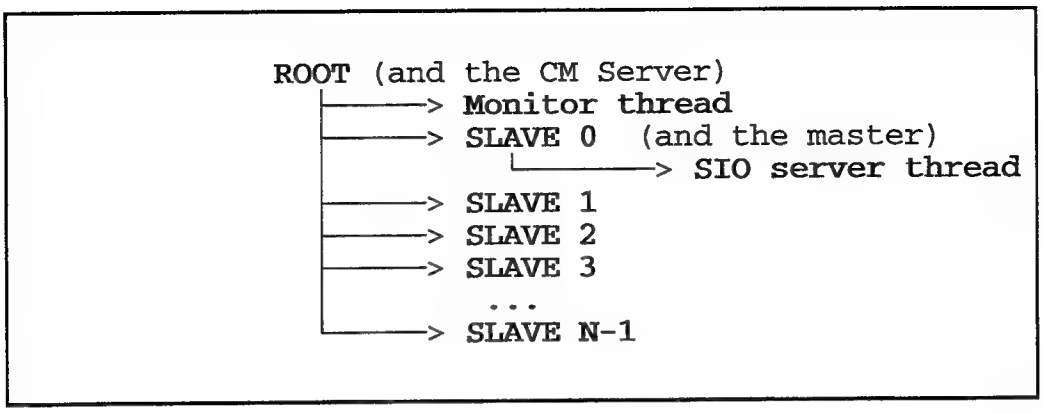

Figure 4-1. Simulator processes and threads.

implemented with the help of pipes because the pipe automatically enforces the required synchronization between communicating processes.

4.3 PROCESS AND THREAD STRUCTURE

The root is the first process initiated by running the simulator. The root creates a child process for each slave; the application program runs in the child processes. The root also creates a slave monitor thread. During execution of the application, the root runs the CM server. The root does not perform any host functions. The first slave process (slave 0) creates a special thread that runs the SIO server. This process is also responsible for other host functions, such as console I/O. Figure 4-1 shows the relationships between the various processes and threads.

4.3.1 The Root Process

This is a list of tasks done by the root. They are performed sequentially, so this serves as a reasonable flowchart of the simulator main program.

1. Establish the exit function (see Section 4.3.3)
2. Interpret the command line arguments
3. Create CM semaphores
4. Allocate shared memory for CM pipes
5. Create CM pipes

6. Pack a few crucial items of information into the "environment string" to be passed on to the slave child processes

7. Create the child processes for all the slaves not in the debug mode

8. Create the debug sessions for all the slaves in the debug mode

9. Create the slave monitor thread

10. Execute the CM server while the application processes execute

11. Wait for the child processes (and debug sessions) to terminate

4.3.2 The Master and Host

Master and host services that must be simulated relate to I/O: console input and output, opening and writing to files on the host computer, etc. In the simulator, these tasks are assigned to the slave 0 process. There is no separate master because there is no broadcasting of instructions to the slave processes. There is no separate host because the slave 0 process has direct access to the physical PC peripherals: console and disks. The standard I/O library has been replaced with functions that are executed if the slave number is 0 but not executed in other slaves. These functions send their return values to all the slaves (via pipes), so the slaves all proceed as if they had received a broadcast return value from the master.

4.3.3 The Slave Monitor

The root process spawns a special thread which effectively monitors the slaves and aborts the entire simulation if a problem causes any one of the slave processes to terminate prematurely. This commonly occurs when the user breaks out of a running simulation because the Ctrl-Break signal is intercepted by only one of the slave processes. When any slave process ends, it posts a semaphore that is detected by the slave monitor. The monitor has access to all of the process IDs and can kill the processes to terminate the simulation.

4.3.4 The SIO Server

Slave process number zero spawns a special thread that executes the "SIO server." This code handles reads and writes of data between the slaves' memory and disk files. The SIO server accesses slave data via pipes.

4.3.5 The CM Server

While the slave processes are running, the root process executes a piece of code called the "CM server." This code responds to any requests for interslave communication encountered in the application program by establishing pipe connections between the appropriate slave processes. To carry out the transfer, these pipes connect source slave to destination slave.

4.3.6 Task Synchronization

Many of the details of how the simulation progresses are handled by the operating system. The questions of scheduling the slave processes for execution, timeslicing, synchronization of piped data, etc., are all under the control of OS/2. Thus, it is not possible to specify the order of execution of the different slave processes. It is important to understand this when viewing the progress of one or more slaves in debug mode. The proper synchronization of interslave transfers at the algorithmic level is enforced by the placement of communication functions in the application code. The simulator faithfully reproduces the result of interslave communication for all legal MeshSP programs.

4.4 SIMULATION OF INTERSLAVE COMMUNICATION

Interslave communication presents the most complex situation for the simulator. Briefly, the application program (slave process) receives the TCB and sends it to the CM server. The CM server interprets the path and leg information and establishes the connections between slaves. These connections are required to transmit data from source to destination; the simulator does not actually pass data through all the intervening slaves as does the MeshSP hardware. The slave process takes the connection information in the form of pipe handles. Data are written to and read from pipes to effect the transfer. Additional complication arises from the need to simulate constant registers and broadcast mode. Synchronization of the CM server and the slaves is an important issue. We concentrate on the OS/2 processes shown in Figure 4-2.

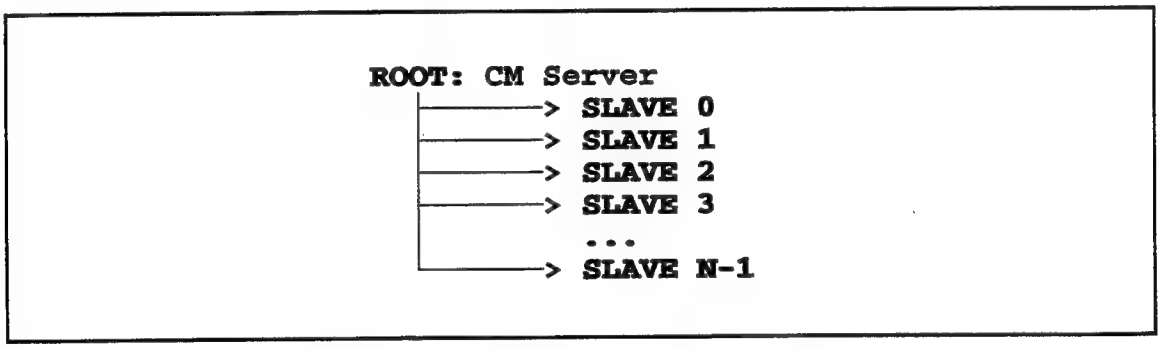

Figure 4-2. Simulator processes involved in CM.

The application code requests a transfer by calling CM(). The transfer is carried out by either CM() or WaitCM(), depending on whether the communication mode is early or late. The CM server, running in parallel, helps by making the required pipe connections.

4.4.1 CM Server Running in Root Process

The pseudocode in Figure 4-3 below indicates the actions taken by the CM server as it defines the connections required to simulate interslave communication. The words break and continue are used in the same sense as the C language keywords.

```
Loop over TCBs
{
      read path information
      if end of program,  break
      if end of TCB,  continue
      SYNCHRONIZE with all the slaves
      loop over one-slave shifts
      {
            get transmit and receive directions for this shift.
            get a constant from the constant register, if needed.
            set "done" flag if all shifts have been traversed.
            if not done
            {
                  make sure transmit and receive directions match
                  determine destination slave for this shift
                  pass along the constant from constant register
            }
            update array of destination handles for each slave
            if this is the last shift or broadcast mode
            {
                  SYNCHRONIZE with all the slaves
                  write destination handles to slaves via pipes
            }
      if done, break
      }
}
```

Figure 4-3. CM server.

To understand this algorithm, it is crucial to understand the following bit of indirection: the server uses interprocess pipes to send pipe handles to the slaves. Each "source" slave receives the handle of a pipe to be used when sending data to its "destination" slave. Each slave has a CM read pipe handle and a CM write pipe handle assigned to it. The pipe is defined in the simulator main program as a connection between these handles. For a transfer from slave A to slave B, the server passes the B write handle to A via the A read pipe. Slave A then affects the CM transfer by writing its data to the B write pipe. Slave A retrieves its new data from its own read pipe.

4.4.2 CM in a Slave Process

The pseudocode in Figure 4-4 below indicates the actions taken by each slave process in the interslave communication routine.

```
loop over TCBs in a chain
{
      send the TCB to the server
      if the chain pointer is FINI,  break
      SYNCHRONIZE with server
      get leg lengths
      loop over the "stores" (normally one; more for broadcast)
      {
            SYNCHRONIZE with server
            read CM destination write handle via CM read handle
            write data, according to TCB, via CM dest. write handle
            read data, according to TCB, via CM read handle
      }
}
```

Figure 4-4. Slave-process actions during communications.

4.5 COMMUNICATION MODES

The simulator does not reproduce the interslave communication details that proceed in the MeshSP hardware. In fact, the MeshSP communications are almost always concurrent with processor computational activity, while the simulator necessarily executes these tasks sequentially. There is some freedom concerning the order of operations that gives rise to two different "modes" for simulator communication. Any legal MeshSP program must be insensitive to this ordering. The two modes are provided to help identify algorithm errors involving the ordering of computation and interslave communication.

Communications are initiated with the CM() function and then proceed at any rate, constrained only by the requirement that all communications will be completed before the next WAITCM() function is completed. In contrast, the simulator carries out all communication within either the CM() or the WAITCM() function. The former case is referred to as "early communication" and the latter is "late communication." These are, in some sense, extreme cases that bracket the situation in hardware, where the bulk of communication occurs at points between the execution of these two functions. Figure 4-5 indicates the difference between the two modes.

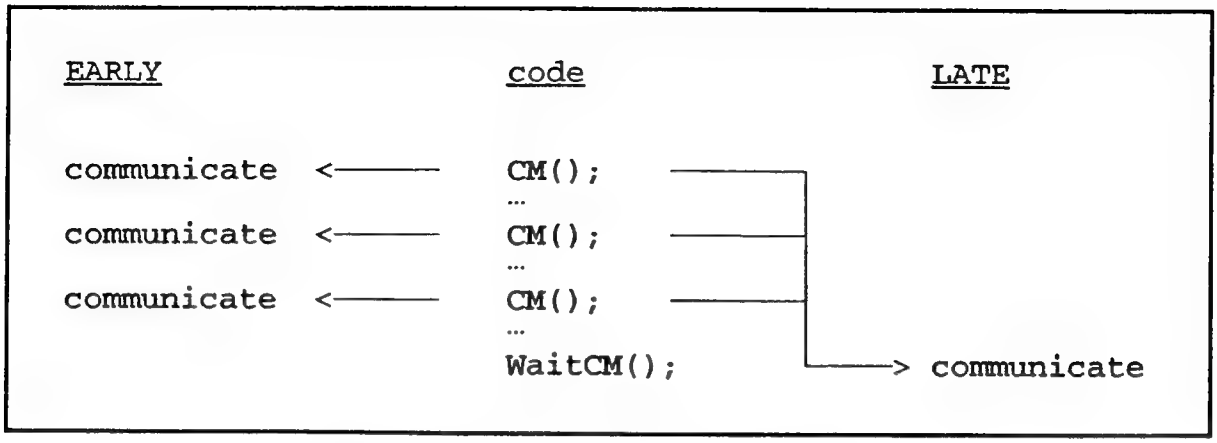

Figure 4-5. Early and late communication modes.

5. ALGORITHMIC EXAMPLES

As emphasized in Section 3, MeshSP algorithm development requires the explicit partitioning of data across slaves, as well as the explicit specification of interslave communication and I/O. The material in this section will illustrate this process by describing a diverse collection of sample problems.

5.1 TREND AND MEAN REMOVAL

Before estimating the power spectral density (PSD) of some data, it is usually necessary to compute and remove the global mean and linear trend.

$$D(X,Y) \rightarrow D(X,Y) - \langle D \rangle - \frac{X \langle XD \rangle}{\langle X^2 \rangle} - \frac{Y \langle YD \rangle}{\langle Y^2 \rangle} \tag{1}$$

where D(X,Y) is the data field, $\langle\ \rangle$ indicates averaging, and X and Y are coordinates of the data points relative to the *global* center. If the coordinates are expressed in units such adjacent points differ by 1, it may be shown that: $\langle X^2 \rangle = \langle Y^2 \rangle = (N^2-1)/12$. To apply Equation (1), the global zeroth and first moments must be formed.

Let X_i and Y_i be the coordinates of the center of the data patch assigned to slave i, and let x and y be coordinates relative to the slave data center. Then

$$\langle XD \rangle = \frac{1}{N} \sum_{i=1}^{N} (\langle xD \rangle_i + X_i \langle D \rangle_i) \tag{2}$$

$$\langle YD \rangle = \frac{1}{N} \sum_{i=1}^{N} (\langle yD \rangle_i + Y_i \langle D \rangle_i) \tag{3}$$

where N is the array size (64 for MeshSP-1). These equations express the global moments in terms of the local moments computed relative to the slave data center.

If the number of data elements per slave is large, the task of combining the local moments is a minor part of the data conditioning algorithm. It may be efficiently accomplished by first forming partial sums in the x-direction and then combining the partial sums in the y-direction. To compute the partial sums, first spread the local moments in the x-direction. This provides each slave with the values of the local moments in the other slaves in the same row. These values are then combined to form three partial sums (every slave in a given row having the same value for each partial sum). The partial sums are then spread in the y-direction, allowing each slave to form the final sums.

The communication cost/moment for the data spreading for 8×8 slaves is

2 directions × 7 shifts × 4 cycles/shift = 56 cycles .

The arithmetic cost is

2 directions × 8 cycles = 16 cycles .

If forming more than three global sums, a more efficient algorithm is available. In fact, if the number of sums is a multiple of the array size, the global sums can be formed with *no computational inefficiency* (albeit with significant communication overhead).

Let A be an array of N elements, i.e., the number of data elements within each slave equals the number of slaves in the array. By first applying the `Permute_X()` operation and then the `Permute_Y()` operation, the N corresponding elements of array A in each slave may be brought together in a single slave. Thus, A[0] in each slave may be brought into slave 0, etc. Once grouped together in this way, a sequence of N arithmetic operations (on nonredundant data in the various slaves) serves to simultaneously form all N required sums. The final results may then be redistributed to the individual slaves by a variety of means, e.g., the `SpreadX()` function followed by `SpreadY()`. Each permutation operation involves an average distance of two cells or eight cycles. Redistributing the final sums involves a cost of four cycles per word. Thus, the cost for each of the N sums is given by

Communication	=	20 cycles ;
Computation	=	1 cycle ;
Total	=	21 cycles .

5.2 SEGMENTED CONVOLUTION

A common operation in signal and image processing consists of convolution with a finite kernel. This may be accomplished on the MeshSP by distributing the data across the processor array, and then simultaneously convolving the data in each slave with a similar filter kernel.

The convolution may be performed directly in real space or via multiplication in Fourier space. In either case, each slave must access data from surrounding slaves. In general, the area to be imported from neighboring slaves must be large enough to encompass the convolution kernel (or at least the region where it is significantly large). This allows the convolved segments to be pieced together, yielding a result that is equivalent to convolving the original, unsegmented data with the filter kernel. This communication may be performed either before convolution (overlap and save) or afterward (overlap and add). The situation is illustrated in Figure 5-1.

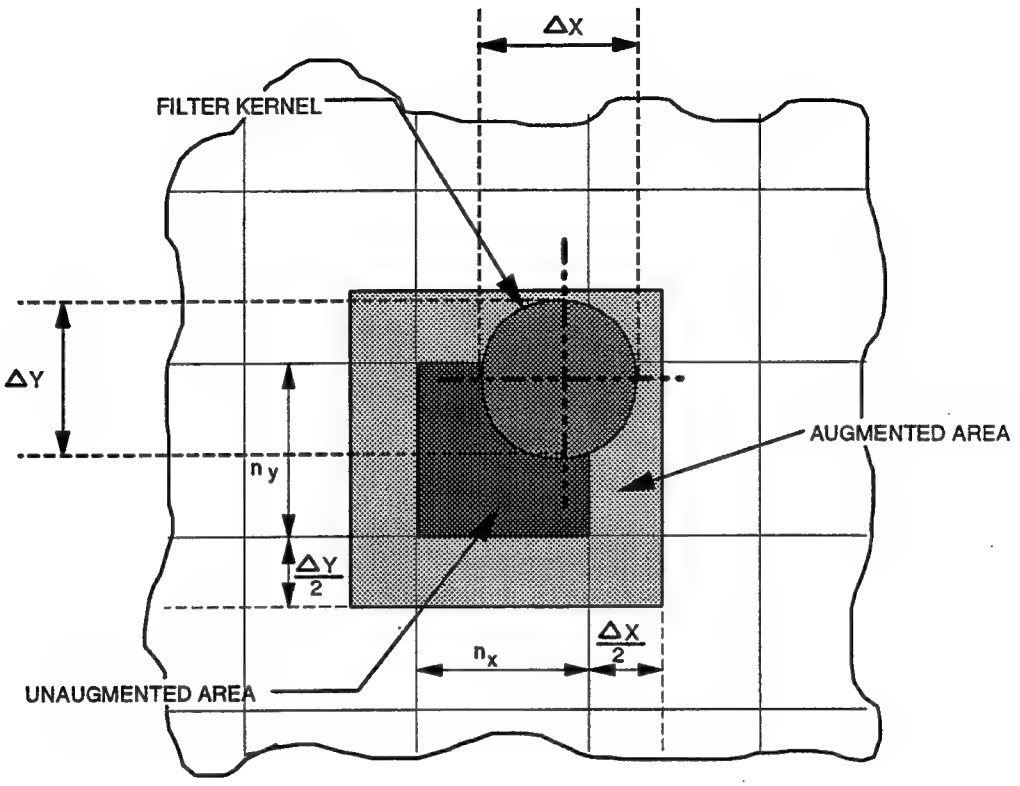

Figure 5-1. Segmented convolution.

In the **overlap-and-save** algorithm, the internal data of each slave are first augmented with data from surrounding slaves. The augmented data of the various slaves are then simultaneously convolved with the filter kernel. If the convolution is done in real space, only the values at points within the original, unaugmented domain need be computed. Alternatively, the convolution may be performed by transforming the augmented data to Fourier space, multiplying by the transform of the convolution kernel, followed by inverse transformation back to real space. This results in the *cyclic* convolution of the data with the filter kernel. Only the interior points of the resultant field are valid, and the points outside the unaugmented region are simply discarded.

The **overlap-and-add** algorithm begins with data from the unaugmented area only. In real space, only interior (unaugmented) points are used as *source points* for the finite convolution. In Fourier space, the data in the augmentation zone are explicitly zeroed before being transformed. In either case, the full convolution

is completed by importing into each slave the overlapping partial results from neighboring slaves and summing them with the internal result to form the full convolution.

```
Augment(g, f, nx, ny, Δx/2, Δy/2)
Loop over x,y, points inside unaugmented domain
{
      g(x,y) = 0
      Loop over x', y', points inside kernel support space
      {
            g(x,y) = g(x,y) + h(x',y') * f(x+x', y+y')
      }
}
```

Figure 5-2. Overlap-and-save algorithm.

High-level functions `Augment()` and `Excise()` (Section 3.4) have been provided to perform the necessary data communication and trimming for the overlap-and-save method.

Let the filter kernel be h(x,y) with support within a rectangle whose x and y dimensions are Δx and Δy. Let nx and ny be the dimensions of f(x,y), the unaugmented data assigned to each slave, while $Nx=nx+\Delta x$ and $Ny=ny+\Delta y$ are the dimensions of the augmented data. Let g(x,y) be the result of convolving the data with the filter.

5.2.1 Real-Space Segmented Convolution by the Overlap-and-Save Algorithm

For real-space convulution we require that Δx and Δy be even. The real space overlap-and-save algorithm is represented by the pseudocode in Figure 5-2. Note that there is no *computational* penalty for this parallel decomposition. No numerical operation is duplicated among the slaves. The computational speedup is simply the ratio of the total processed area to the area assigned each slave.

There is a communication penalty involved in gathering data from surrounding slaves which depends on the distance from which the data must be gathered. For simplicity consider a kernel limited in extent by There is a communication penalty involved in gathering data from surrounding slaves which depends on the distance from which the data must be gathered. For simplicity consider a kernal limited in extent by $\Delta x < 2n_x$ and $\Delta y < 2n_y$. It takes four processor cycles to communicate a single 32-bit word from a nearest neighbor, eight cycles from a next nearest neighbor, etc. Referring to Figure 5-1, one may determine the total communication cost (in cycles) to be $4(n_x\Delta_y + n_y\Delta_x + 2\Delta_x\Delta_y)$. If the filter kernel fills the rectangle, $\Delta_x\Delta_y$ the single cycle multiply-accumulate operation results in a computational cost of $n_x n_y \Delta_x \Delta_y$. The ratio of communication to computation is then

$$\frac{communication}{computation} \cdot 4\left[\frac{1}{n_x\Delta_y} + \frac{1}{n_y\Delta_x} + \frac{1}{n_x n_y}\right] \tag{4}$$

For virtually all reasonable sizes of data and filter this ratio is small compared with unity and computation dominates communication.

5.2.2 Fourier-Space Segmented Convolution by the Overlap-and-Save Algorithm

To apply Fourier-space convolution the size of the augmented region cannot be arbitrarily selected; it must match one of the transform lengths supported by the FFT library. The MeshSP library supports lengths of the form 2^n and 3×2^n, e.g., 16, 24, 32, 48, 64, 96, etc. The Fourier-space overlap-and-save convolution is given in Figure 5-3.

```
Augment(g, f,  nx, ny, Δx/2, Δy/2)
FFT2D(g, Nx, Ny)
Loop over all points kx, ky in Fourier-space
{
      g(kx, ky) = g(kx, ky) * H(kx, ky)
}
IFFT2D(g, Nx, Ny)
Excise(SET, f, g, nx, ny, Δx/2, Δy/2)
```

Figure 5-3. Fourier space overlap-and-save convolution.

Here, `FFT2D()` is a two-dimensional FFT routine which transforms real data to Hermitian data, while IFFT is its inverse. H(kx, ky) is the (appropriately scaled) transform of h(x,y).

This algorithm involves a round-trip real-to-Hermitian FFT plus a single complex multiply over *half* of Fourier space (Hermitian symmetry). In the SHARC, the time to perform a full complex FFT is approximately $2N \log_2 N$ cycles, and a real to Hermitian half that, while a complex multiply takes four cycles. The total computational cost is then: $2N_xN_y[1 + \log_2(N_xN_y)]$. In general, real-space convolution is economical only for filter kernels that are sufficiently compact that $\Delta x\Delta y < [1 + 2\ln_2((n_x+\Delta_x)(n_y+\Delta_y))]$. For example, if the data were distributed so that each slave processed a 64×64 cell portion, the first available size for the FFT would be 96×96 cells. Fourier-space convolution thus costs $96^2[1+2\log_2(96^2)]$ cycles. On the other hand, real-space convolution costs $64^2\Delta_x\Delta_y$ cycles. The crossover point occurs when $\Delta_x\Delta_y = 62$. In this example, real-space convolution is faster than Fourier-space convolution for kernels smaller than 62, while for larger kernels the Fourier-space algorithm is preferable.

It should be pointed out that this simplest form of Fourier-space convolution involves a greater number of FFTs than is absolutely necessary. Because augmentation provides neighboring slaves with redundant data, transforming these data involves avoidable computation. Here, we may exploit the fact that because the original data are real, their transforms are Hermitian. This symmetry allows an entire transformed row, for example, to be reconstructed from essentially half the transformed points. The alternative algorithm is given in Figure 5-4.

```
Augment in x-direction
Row transform in x-direction
      (ny real-to-Hermitian transforms of length Nx)
Augment in y direction
Column transform in y-direction
      (Nx/2 full complex transforms of length Ny)
```

Figure 5-4. A faster two-dimensional FFT.

The computational cost of the modified algorithm, relative to the unmodified algorithm, is then

$$relative\ computational\ cost = \frac{1 + \frac{n_y}{N_y}\cdot\frac{\log(N_x)}{\log(N_y)}}{1 + \frac{\log(N_x)}{\log(N_y)}} \qquad (5)$$

For example, if Nx = Ny = 2nx = 2ny (square array augmented to twice its linear dimension), the relative cost is 3/4, i.e., a 25% saving in computation. Although the savings may be worthwhile, it does introduce an element of coding complexity because the two-dimensional augmentation and FFT library routines cannot be used whole but must be split into parts and interwoven.

5.3 SPATIAL AVERAGING OF EXTENDED DATA

A major advantage of the MeshSP architecture is its scalability. It is possible to construct quite large processor arrays that can process distributed data via such segmentation algorithms as described above. A common feature of data adaptive extended processing is the construction of statistics (mean, max, min, etc.) over neighborhoods surrounding each slave.

A straightforward (but inefficient) proceedure is to first gather surrounding data via the `Augment()` function and then sum them in place. For reasons of symmetry let N, the size of the averaging region, be an odd number of slaves. The staged augmentation algorithm illustrated in Figure 3-3 involves a communication cost of 1 slave step (four cycles) per imported word *regardless of the size of the augmentation area.* Thus, augmenting a single quantity over an $N \times N$ slave region costs $4N^2$ cycles. The cost of summation must be added, another N^2 cycles. For example, if N=9 slaves,

Communication:	81 shifts × 4 cycles/shift	=	324 cycles
Computation:			81 cycles

A more efficient algorithm, **divide and conquer**, can significantly reduce these costs. It is expecially suitable if N is a power of 2 or 3. Suppose once again that $N = 3^2 = 9$. To form a sum over the 81 slave regions centered on a given slave, first import into each slave the data values from the two nearest neighbors in the x-direction. The internal value is added to the imported values to form a partial sum. The process is then repeated with the two nearest neighbors in the y-direction. At this point each slave has computed the sum of the data in the 3×3 slave area centered on itself. The entire sequence is then repeated with the importation of data from a distance of 3 slaves in each direction (see Figure 5-5).

The costs of the divide-and-conquer algorithm for 9×9 averaging are

Communication:	1st stage	4 shifts x 4 cycles/shift	=	16 cycles
	2nd stage	12 shifts x 4 cycles/shift	=	48 cycles
	Total			64 cycles
Computation:	1st stage			6 cycles
	2nd stage			6 cycles
	Total			12 cycles

Note that these costs ignore the overhead of initiating communication. Simply moving the TCB into the communication hardware costs 15 cycles, and additional cycles are needed for the operations associated with the CM control structure (Section 3.3). However, when one is interested in the statistics of a substantial number of quantities (for example, the elements of a two-dimensional power spectral density) this overhead becomes a small part of the total cost.

Another potential drawback to the divide-and-conquer technique is the back and forth alternation of communication and computation at each stage. This threatens to make concurrency with other computation difficult during the various CM segments. The problem is avoided by programming the computational parts of the divide and conquer as a function to be called by the CM interrupt service routine via a pointer in INT_VECT_TABLE (see Section 3.3). This causes the divide-and-conquer additions (or other computation) to be performed as soon as the communication for each stage is complete, while allowing for full concurrency with an unrelated computational task during the lengthy communication segments.

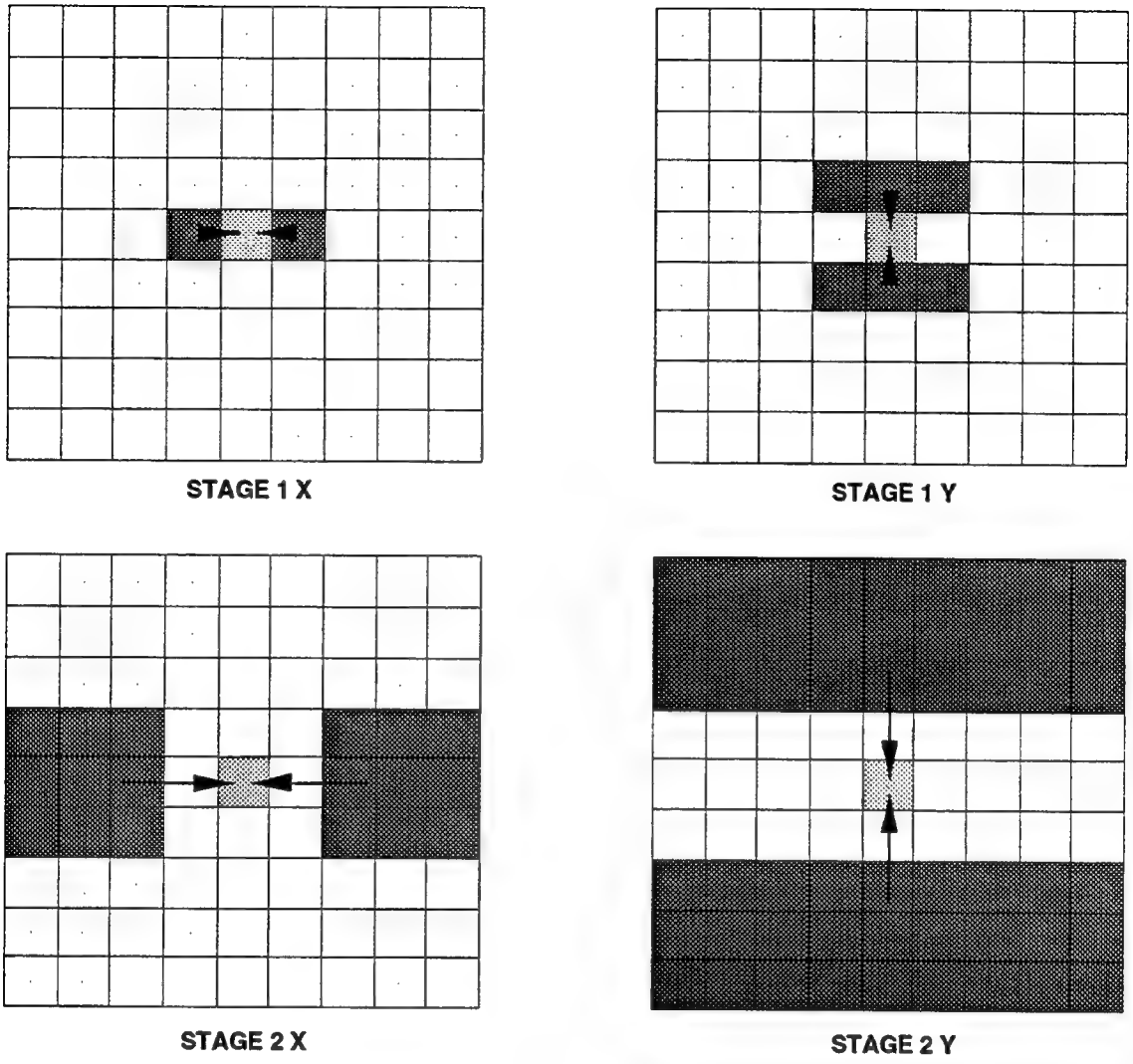

Figure 5-5. 9×9 ternary divide and conquer.

5.4 GLOBAL TWO-DIMENSIONAL FFT

The MeshSP can efficiently perform Fourier transformation of data distributed across the entire processor array. Because each point of the original data affects all points of the transformed data, it is clear that interslave communication is a major component of the process. A straightforward algorithm for performing a global two-dimensional FFT is

1. Redistribute the data so that each complete row is placed in a single slave

2. Transform the rearranged rows by a series of within-slave one-dimensional FFTs
3. Restore the transformed data by the inverse of the row rearrangement
4. Redistribute the data so that each column is placed in a single slave
5. Transform the rearranged columns by a series of within-slave one-dimensional FFTs
6. Restore the fully transformed data by the inverse of the column rearrangement

5.4.1 Data Shuffling

The interprocessor communications required to implement this general procedure are based on the functions `Permute_X()` and `Permute_Y()`, described in Section 3.4. Their operation may be examined in detail. For simplicity we assume the processor array to be square, with

ARRAY_X_DIM = ARRAY_Y_DIM = AD = 8, the size of MeshSP-1.

Let the complex data array have dimensions NX and NY, each a multiple of AD:

NX = AD*Nx

NY = AD*Ny

The complex data array assigned to each slave has dimensions Nx and Ny.

Complex quantities are stored with real and imaginary parts in immediately adjacent memory locations. Similarly, complex row elements are stored in adjacent pairs of locations. In packing a complex row one is free to treat it as if it were a real row with twice the number of points (see Figure 5-6).

The situation is more complicated for columns. As shown by the right-hand poriton of the figure, the complex columns cannot be considered to consist of eight interdigitated *two-dimensional* subarrays. The problem is that *three* different address increments are needed to describe each subarray: 1 (real part to imaginary part), 2 NX/8 (between adjacent elements in the same column), and 16 (between an element in a given column and its counterpart in the next column). Note, however, that the real part of the data and the imaginary part of the data separately form interdigitated two-dimensional subarrays. Thus, the process of packing each column into a single slave must be accomplished by applying *two* permutations: one for the real part and one for the imaginary part of the complex data. The `FFT()` function expects one-dimensional data with real and imaginary parts in adjacent locations. Therefore, the column packing transforms data from the format designated as column addressing to the format designated as row addressing in the figure.

The details of programming this complex communication may be understood by examining the code that prepares the TCBs.

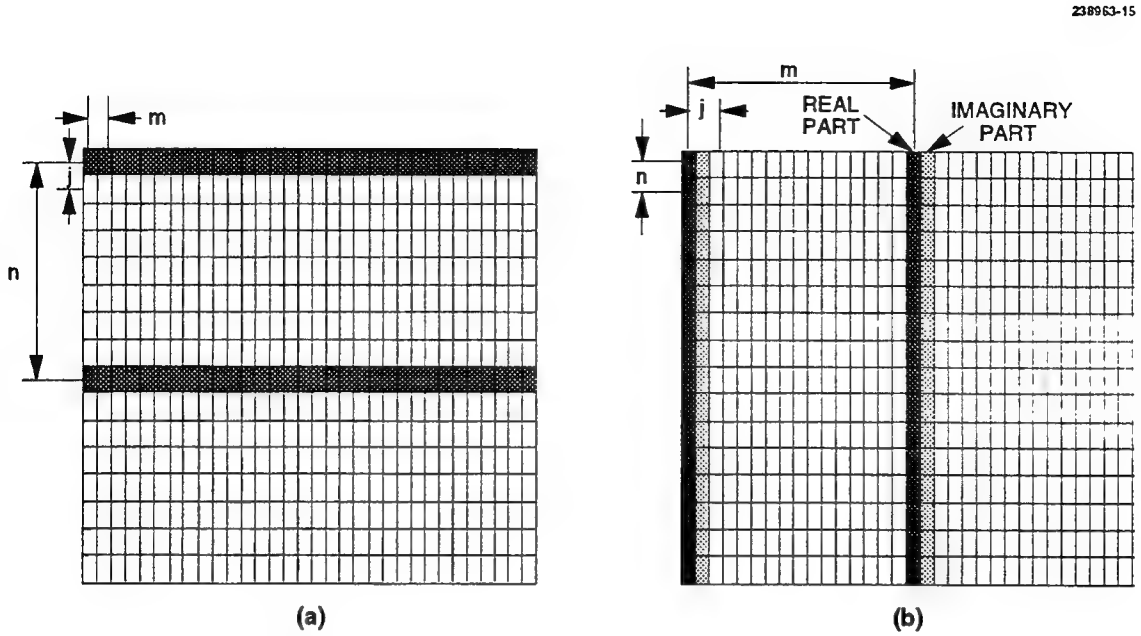

Figure 5-6. Addressing formats for row and column packing.

To prepare for the row packing the dma structures must first be constructed. They specify the arrangement of data for the original row-addressed array and for the FFT buffer. The two arrangements are identical; only the base addresses differ.

```
MkDmaC( &row dma,   data,   Nx, 0,  1, Nx,  0, AD,  Ny/AD );
MkDmaC( &buf dma, buffer,   Nx, 0,  1, Nx,  0, AD,  Ny/AD );
```

Next one must construct the dma structures for the real and imaginary parts of the column addressed data and FFT buffer.

```
MkDmaY( &r col dma, data,   2*Nx, 0, 2*AD, Nx/AD, 0,  1, Ny );
MkDmaY( &i col dma, data,   2*Nx, 1, 2*AD, Nx/AD, 0,  1, Ny );
```

The complex buffer dma created above cannot be used directly because it combines real and imaginary parts. Two structures are required, one for the real part and one for the imaginary part.

```
MkDma( &r_buf_dma, buffer, 2*Nx, 0,   2, Nx, 0, AD,  Ny/AD );
MkDma( &i_buf_dma, buffer, 2*Nx, 1,   2, Nx, 0, AD,  Ny/AD );
```

Having specified the form of the required subarrays, one may then create the required TCB to perform the permutations. Each permutation involves as many different shifts as processor rows or columns, which is eight in this example. For the global FFT we require six different transformations, as shown in Figure 5-7.

```
1. pack complex rows
   (fft rows)
2. unpack complex rows
3. pack real part of columns
4. pack imaginary part of columns
   (fft columns)
5. unpack real part of columns
6. unpack imaginary part of columns
```

Figure 5-7. Possible transformations for the global FFT.

This requires an array of $6 \times 8 = 48$ TCBs. In C notation

```
cm_tcb  TCB[6*ARRAY_X_DIM];
```

Although there are 48 TCBs, they may be linked to form just three chains: one to prepare for the row ffts, one to prepare for the column ffts, and one to restore the data to its original arrangement. First, construct the TCBs to pack and unpack the rows.

```
Permute_X( TCB,      +1,   buf_dma, 2*Nx, row_dma, 2*Nx );
Permute_X( TCB + AD, -1,   buf_dma, 2*Nx, row_dma, 2*Nx );
```

Next construct the TCBs to pack and unpack the real part of the columns.

```
Permute_Y( TCB+2*AD, +1, r_buf_dma, 2*Nx, r_col_dma, 2 );
Permute_Y( TCB+4*AD, -1, r_buf_dma, 2*Nx, r_col_dma, 2 );
```

Next, construct the TCBs to pack and unpack the imaginary parts of the columns.

```
Permute_Y( TCB+3*AD, +1, i_buf_dma, 2*Nx, i_col_dma, 2 );
Permute_Y( TCB+5*AD, -1, i_buf_dma, 2*Nx, i_col_dma, 2 );
```

Finally, link up the chains that may be combined.

```
LinkCM( TCB+2*AD-1,TCB+2*AD );/*unpack row -> pack real column          */
LinkCM( TCB+3*AD-1,TCB+3*AD );/*pack real column -> pack imag. column   */
LinkCM( TCB+5*AD-1,TCB+5*AD );/*unpack real column ->unpack imag.column */
```

Having created these TCB chains, the entire global fft consists of the following code.

```
CM( HIGH, TCB );      /*    pack rows                    */
WaitCM( TCB );
for(i=0; i<n; i++)    /*    transform n rows             */
        FFT( buffer+i*NX, NX, isign );
```

```
CM( HIGH, TCB+AD );  /*     unpack rows, pack columns     */
WaitCM( TCB+AD );
for( i=0; i<n; i++ ) /*     transform n columns           */
       FFT( buffer + i*NY, NY, isign );
CM( HIGH, TCB+4*AD );/*     unpack columns                */
WaitCM( TCB+4*AD );
```

5.4.2 Timing and Throughput Considerations

The division of data among slaves provides full *computational* efficiency. No cycles are wasted as each slave is fully occupied in nonredundant computation. The communication process does, however, exact a significant price.

By virtue of toroidal connectivity, the *greatest* distance by which a data item need be moved in either a row or column permutation is half the array dimension, or AD/2. The average distance is half that, or AD/4 slaves. If the cost of the zero shift intraslave transfer is included this average cost is increased by 1/AD. This correction is small and will be ignored. Because four cycles are required to shift each 32-bit real quantity, the average communication cost/real word is AD cycles.

The global two-dimensional transform described above involves four transfers (row pack, row unpack, column pack, and column unpack) for the real and imaginary part of each complex data point. For a transform of N complex points (N/AD^2 per slave) the total communication cost is therefore $4*2*AD*N/AD^2$ cycles. Because a *single* SHARC performs an N-point complex FFT in $2N*\log_2(N)$ cycles, the time for the array computation is $2N*\log_2(N)$ cycles/AD^2 cycles. Thus, the ratio of communication to computation is: $4*AD/\log_2(N)$.

For example, a 1024×1024 ($N=2^{20}$) point transform on an 8×8 processor array (AD=8) takes about 0.65×10^6 cycles (16 ms) for computation and 1.0×10^6 cycles (25 ms) for communication. If computation and communication are performed sequentially, the total time for the FFT is the sum, or 41 ms. By proper pipelining, it is usually possible to hide much or all of the faster process (computation) behind the slower process (communication). This is especially easy if a number of such transforms are to be performed in sequence. In that case the total time is the greater of the communication and computation times. For the example cited, the slowdown factor associated with data distribution would then be 16 ms/25 ms=0.64.

5.5 LARGE ONE-DIMENSIONAL FFT

The global FFT of Section 5.4 may be used as the basis for a one-dimensional transform of a large number of points distributed across the processor array. The algorithm used is the first step in a standard derivation of the fast Fourier transform. Let A[t], (t = 0,1,..,N-1) be a complex vector of dimension N. Its transform is defined as below.

$$\overline{A}(f) = \sum_{k=0}^{N-1} e^{\frac{i2\pi ft}{N}} A(t) \tag{6}$$

For simplicity we assume the processor array has dimensions 8×8.

Let

N	=	NX*NY, where NX and NY are each multiples of 8 and of the form 2^n or $3x2^n$.					
t	=	NY*x + y,	x	=	0 .. NX-1	y	= 0 .. NY-1
f	=	$k_x + NX*k_y$	k_x	=	0 .. NX-1	k_y	= 0 .. NY-1

Note that while the row-column representation for the frequency index *f* follows the standard convention of the *x* index varying more rapidly than the *y* index, that is not the case for the time index *t*. The original data are to be placed in the two-dimensional array so that adjacent values are arranged along columns, not rows. With this representation,

$$\overline{A}(k_x + NXk_y) = \sum_{y=0}^{NY-1} e^{i\frac{2\pi yk_y}{NY}} e^{i\frac{2\pi yk_x}{N}} \sum_{x=0}^{NX-1} e^{i\frac{2\pi xk_x}{NX}} A(NYx+y) \tag{7}$$

This equation may be interpreted as follows.

1. FFT in *x*-direction (rows)
2. Multiply result of first step by a phase factor

$$e^{i\frac{2\pi yk_x}{N}} \tag{8}$$

3. FFT in the *y*-direction (columns)

This process is essentially equivalent to the global two-dimensional FFT, which requires the additional step of multiplication by a slave and index dependent phase factor. For repeated FFTs these factors may be computed once and then stored for later use. In that case the timing considerations are essentially the same as for the global two-dimensional FFT.

5.6 SYSTEMS OF LINEAR EQUATIONS

The importance of linear algebraic equations in science and engineering makes the parallel solution of such problems the subject of much recent research [3,4]. The difficulty in parallelizing these problems for a SIMD architecture revolves around communication between processor elements. As in the Fourier transform problem, each element of the answer is dependent on every data item in the original problem. Solution requires frequent access to data which are likely to be widely dispersed across many PEs. In the most general cases, known as "dense matrix" problems, the computational complexity is of order N^3, where N is the number of equations in the linear system. On the other hand, the communication load would be expected to grow in proportion to the number of coefficients per processor (order N^2). One therefore expects that a large problem will be better suited to SIMD parallelization than a small one. Although a survey of parallel linear algebra techniques is beyond the scope of this report, we will describe some example algorithms for solving linear equations and discuss the implementation efficiency on the MeshSP.

Consider the standard problem involving a set of N simultaneous linear equations and N unknowns x_i. This is expressed by the matrix equation:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{9}$$

where **A** is an $N \times N$ matrix of coefficients, **x** is the vector of unknowns, **b** is the right-hand side vector, and the dot indicates matrix multiplication. One may solve for a collection of right-hand sides

$$\mathbf{A} \cdot \mathbf{X} = \mathbf{B} \tag{10}$$

where **B** is a set of right-hand side vectors in the form of a matrix and **X** is a set of solution vectors. If the right-hand side vectors form an $N \times N$ unit matrix, then the solution of

$$\mathbf{A} \cdot \mathbf{X} = \mathbf{I} \tag{11}$$

provides the inverse of matrix **A**.

A well-known technique for the solution of these problems is Gauss-Jordan elimination. As shown below, this algorithm accesses the matrix elements in a regular fashion, which is compatible with SIMD decomposition. For the matrix inversion problem, Gauss-Jordan elimination is about as fast and stable as any other inversion technique[5]. For the situation where a set of linear equations must be solved without requiring $\mathbf{A}^{-1}$, there are faster methods. For example, "triangular" Gaussian elimination with backsubstitution is about three times faster on a uniprocessor. This will be discussed Section 5.6.2.

5.6.1 Gauss-Jordan Elimination

The basic algorithm involves the selection of a pivot element (the largest element in a particular column i), normalization of that row by the pivot element (which becomes 1), interchanging rows to place the pivot

element on the diagonal, and subtraction of a multiple of the new i^{th} row from all other rows in such a way as to zero the off-diagonal elements in the i^{th} column. Each of these operations is also performed on matrix **B** using the same numerical weights used for matrix **A**. The result is that the original equations

$$\boldsymbol{A} \cdot \boldsymbol{X} = \boldsymbol{B} \tag{12}$$

are converted to

$$\boldsymbol{I} \cdot \boldsymbol{X} = \boldsymbol{A}^{-1} \cdot \boldsymbol{B} \tag{13}$$

To be specific, the serial algorithm is given in Figure 5-8 below in pseudocode. This includes partial pivoting, which is required for numerical stability.

```
/* Loop over the columns */
for (i=0; i<N; i++)

      /* Find the pivot element in column i */
      for (j=i; j<N; j++)
            Find the largest A[j][i] in absolute value
            This becomes the pivot value pv
            The pivot row pr is set to j

      /* Normalize the pivot row in A and B */
      for (ii=0; ii<N; ii++)
      divide A[pr][ii] by pv
            divide B[pr][ii] by pv

      /* Interchange row pr with row i in A and B */
      for (ii=0; ii<N; ii++)
            swap A[pr][ii] with A[i][ii]
            swap B[pr][ii] with B[i][ii]

      /* Eliminate the pivot row pr from other rows in A and B */
      for (j=0; j<N; j++)
            for (ii=0; ii<N; ii++)
                  subtract A[j][i]*A[pr][ii] from A[j][ii]
                  subtract A[j][i]*B[pr][ii] from B[j][ii]
```

Figure 5-8. Serial Gauss-Jordan elimination.

A large system of linear equations may be distributed across the MeshSP array by placing one column of the problem in each PE. This is very compatible with row exchange and row subtraction because each processor holds the data needed to do those operations. However, the weights required for row subtraction

must be communicated from the processor with the pivot column. The columns are distributed evenly across the processors. For example, the solution of 60 equations and 60 unknowns with four different right-hand sides is a 64-column problem that fits in an 8×8 MeshSP array. The inversion of a 64×64 element matrix is a 128-column problem, where each processor has a column from matrix A and a column from matrix **B** = **I**. There is no restriction on the dimensions of the problem because the problem may be augmented to a multiple of 64 by padding with extra right-hand columns. There is some inefficiency in processor utilization when the number of columns is not an even multiple of the number of Pes.

For concreteness, we focus on the **square matrix inversion** problem, where the matrix width N is a multiple of the number of processors P. This problem is free of the inefficiencies noted so far: (1) Gauss-Jordan elimination provides an efficient solution to the matrix inversion problem, and (2) no column padding is required. At issue will be the capabilities of the MeshSP PE, the inefficiency introduced by the SIMD requirement, and the necessity of communicating intermediate results between PEs.

The column-oriented algorithm suffers some inefficiency in the selection of the pivot. This cannot be parallelized across columns because the selections for the different columns are done sequentially with intervening row elimination operations. In addition, the row elimination requires that each PE contain the normalized pivot column. Therefore, it is straightforward (though inefficient) to communicate the column to all PEs and perform P identical pivot selection and column normalization operations. Becuase the PEs perform the pivot selection on the same data, there are no special SIMD considerations (such as the need for SIMD-compatible branching surrogates). For large problems the row elimination computation overshadows the pivot selection inefficiency.

The communication of the pivot column is efficiently done with the "spreading" functions using the 21060 "broadcast communication" mode. The PE with the pivot column places it in a buffer for communication, while the other PEs fill their corresponding buffers with zeros. Then the buffer elements are communicated along the columns of the MeshSP array with a copy of each communicated item dropped off at each PE along the way. Summation of corresponding elements communicated from all PEs results in the pivot column vector being replicated in each PE.

Figure 5-9 gives our column-oriented SIMD parallel version of the Gauss-Jordan routine. This solves the N×N element matrix inversion problem on an MeshSP array with N PEs.

The generalization to larger arrays is simple: each PE holds several columns of arrays **A** and **B**. For example, by devoting about 32% of the SHARC's internal memory to the storage of **A** and **B**, a 1024×1024 element matrix with an 8×8 MeshSP processor can be inverted. In this problem each PE holds 16 columns of **A** and 16 columns of **B**.

The maximum possible performance for the matrix inversion problem is addressed. We assume the time-critical sections are assembly-coded, and the coding details may vary depending on the problem dimension. The performance benchmarks necessarily reflect the unique capabilities of the 21060 core

processor, as well as the MeshSP SIMD architecture. For example, the 21060 can perform the row elimination inner loop at speeds approaching three clock cycles per pair of points in **A** and **B**. This requires that one array is stored in 21060 on-chip "data memory" in 32-bit words, while the other is stored in on-chip "program memory" consuming 48-bits per word.

```
Download one column of matrix A to each PE
/* Each PE also has one column of the unit matrix B */
/* Loop over the columns */
for (i=0; i<N; i++)
      /* Define a mask that selects the i-th PE */
      z = zero(i-(slave_y + sqrt(N)*slave_x));

      /* Mask out the i-th column, store in C */
      for (j=0; j<N; j++)  C[j]=A[j]*z;

      /* Broadcast the pivot column to all PEs */
      Spread_X ( C );
      Sum the N communicated columns
      Spread_Y ( C );
      Sum the N communicated columns

      /* Find the pivot element in column i */
      for (j=i; j<N; j++)
            Find the largest A[j] in absolute value
            This becomes the pivot value pv
            The pivot row pr is set to j

      /* Normalize the pivot row in A and B */
      divide A[pr] by pv
      divide B[pr] by pv

      /* Interchange row pr with row i in A and B */
      swap A[pr] with A[i]
      swap B[pr] with B[i]

      /* Eliminate the pivot row from other rows in A and B */
      for (j=0; j<N; j++)
            subtract A[i]*C[j] from A[j]
            subtract B[i]*C[j] from B[j]
```

Figure 5-9. SIMD Gauss-Jordan elimination.

Another assumption is that concurrency between computation and communication is exploited by the code whenever possible. The key to obtaining this concurrency is the recognition that pivot columns are selected in a predictable pattern, so the next column may be distributed while the current column is in use for the computationally intensive row-elimination operation. This requires additional row-elimination operations on the communicated column, but this overhead is small when the problem dimension is larger than the number of PEs.

Note that the inversion of large matrices is subject to accumulated round-off error. A common solution to this problem is to refine the answer with an iterative improvement algorithm, usually involving one additional matrix inversion of equal size [5]. The discussion below reflects only a single pass at the inversion.

The cycle counts for the most expensive operations are given below for an MeshSP array of P PEs.

Masking the pivot column	N^2
Communication of the pivot columns	$8N^2\sqrt{P}$
Elimination on the new pivot column	$2N^2$
Finding pivot element	$2N^2$
Elimination on arrays A and B	$3N^3$/ARRAY_SIZE

Timing estimates are shown below for six matrix inversion problems of differing dimension on four different MeshSP arrays. The shaded cells indicate that the problem does not fit within the 512-Kbyte SHARC memory constraint.

Table 1 shows the time to perform the computations for the SIMD inversion algorithm. The times are in seconds and are based on the 40-MHz clock for the SHARC chip. This does not include the communication overhead.

TABLE 1

Computation Time for Matrix Inversion (sec)

Matrix	2×2 PEs	4×4 PEs	8×8 PEs	16×16 PEs
64 × 64	0.005	0.002	0.001	0.001
128 x 128	0.041	0.012	0.005	0.003
256 × 256	0.323	0.087	0.028	0.014
512 × 512	2.55	0.662	0.191	0.073
1024 × 1024	20.3	5.17	1.39	0.447
2048 × 2048	162.0	40.8	10.6	3.04

Table 2 shows the communication time as a fraction of the total computation time. The communication time overwhelms the smaller problems on MeshSP processors with a significant number of PEs.

TABLE 2

Communication Time as a Fraction of Computation

Matrix	2 × 2 PEs	4 × 4 PEs	8 × 8 PEs	16 × 16 PEs
64 × 64	15%	136%	640%	1780%
128 × 128	8%	82%	492%	1710%
256 × 256	4%	45%	326%	1460%
512 × 512	2%	24%	192%	1080%
1024 × 1024	1%	12%	106%	703%
2048 × 2048	1%	6%	55%	413%

Table 3 shows the total time to invert, including the communication time. We assume the use of any available concurrency between communication and computation. Therefore, the total time is less than the sum of time from Tables 1 and 2.

TABLE 3

Total Time for Matrix Inversion (sec)

Matrix	2 × 2 PEs	4 × 4 PEs	8 × 8 PEs	16 × 16 PEs
64 x 64	0.005	0.003	0.006	0.012
128 x 128	0.04	0.012	0.023	0.050
256 x 256	0.32	0.087	0.094	0.198
512 x 512	2.55	0.662	0.370	0.0793
1024 x 1024	20.30	5.17	1.50	3.17
2048 x 2048	162.00	40.80	10.60	12.70

Table 4 gives the SIMD speedup factor. This is defined with respect to a uniprocessor version of the code (no communication or other SIMD overhead) running in a single SHARC chip (with an imagined unlimited on-chip memory).

TABLE 4

Speedup Factor for Matrix Inversion

Matrix	2 × 2 PEs	4 × 4 PEs	8 × 8 PEs	16 × 16 PEs
64 × 64	3.6	7.7	3.4	1.6
128 × 128	3.8	13.2	6.8	3.2
256 × 256	3.9	14.5	13.5	6.4
512 × 512	4.0	15.2	27.0	12.7
1024 × 1024	4.0	15.6	53.9	25.4
2048 × 2048	4.0	15.8	60.8	50.8

Situations where the speedup approaches the number of PEs represent particularly cost-effective matches between the problem and the hardware. For example, the 8×8 MeshSP-1 processor is well suited to inverting matrices with several hundreds of rows.

5.6.2 Gaussian Elimination with Backsubstitution

The computational benefits of this algorithm are most apparent when solving for only one right-hand side vector. Although the inner loops of Gauss-Jordan elimination are executed N^3 times, triangular Gaussian elimination requires only $N^3/3$ iterations, and the subsequent backsubstitution process is proportional to N^2. How well does this parallelize?

The generalization of our column-oriented Gauss-Jordan elimination would have each PE eliminate the pivot row only from rows below the pivot element. This produces an upper-diagonal matrix with about $N^3/2$ executions of the inner loop. If the dimension of the problem is much larger than the number of PEs (and the assignment of PEs to columns is appropriately interleaved), then one can make use of the knowledge that columns to the left of the pivot element do not require row-elimination. This makes it possible to approach uniprocessor efficiency ($N^3/3$ iterations) as N increases. The other great expense, communication of the pivot columns, is cut in half because it is necessary to communicate only to elements below the pivot row.

On the other hand, backsubstitution suffers from additional SIMD inefficiencies. The back-substitution step, executed N times, involves a vector inner-product operation where one vector is a partial row in the main matrix **A** and the other vector is part of the right-hand side **b**. The multiplications can be parallelized by distributing the solution vector **x** and the vector **b** across PEs, but this creates partial sums in each PE that must be communicated to a common point for the computation of the new **x** element. Parallelized backsubstitution remains an N^2 process, making the overall Gaussian elimination algorithm quite efficient for large N. A SIMD algorithm is given in pseudocode in Figure 5-10.

```
/*
      The solution vector x is spread across PEs.
      x[i] and b[i] are assigned to PE number i/P (integer divide).
      Note the interleaved column storage order of matrix A:
      One PE holds A[k*P][j] for all j and k=0..N/P.
      One solution vector element is known: x[N-1] = b[N-1]
*/

Distribute the right-hand side vector b[] across PEs

x[N-1]=b[N-1]

/* Loop over elements of the solution vector x */
for (j=N-2; j<=0; j--)

      /*
            Vector inner product of A*x
            Each CE forms (N-i)/P products
      */
      i = pe_num
      for (k=0; k<(N-j)/P; k++)
        psum = psum + A[k*P+i][j] * x[k*P+i]

      /* Sum the partial sums over to all PEs */
      Spread_X ( psum );
      Sum the communicated items
      Spread_Y ( psum );
      Sum the communicated items, store in sum

      /*  One CE computes the next solution vector element  */
      x[j] = b[j] - sum
```

Figure 5-10. SIMD Gaussian elimination and backsubstitution.

Cycle count estimates for the most expensive operations are given below for a MeshSP array of P PEs.

GAUSSIAN ELIMINATION

Masking the pivot column	$(1/2)N^2$
Communication of the pivot columns	$8N^2 P$
Elimination for the new pivot column	N^2
Finding pivot element	N^2
Elimination for arrays A and B	N^3/P

BACKSUBSTITUTION

Communicate r.h.s. to other PEs	$N\sqrt{P}$
Vector inner product	$N^2/(2P)$
Communicate and add partial sums	$10N\sqrt{P}$

Table 5 shows the total time for the solution of N equations and N unknowns for six different values of N. Table 6 shows the speedup with respect to a comparable uniprocessor version of the algorithm. Shaded entries are prohibited by the SHARC memory size.

TABLE 5

Total Time for Gaussian Elimination (sec)

Equations	2 × 2 PEs	4 × 4 PEs	8 × 8 PEs	16 × 16 PEs
64	0.002	0.001	0.003	0.007
128	0.015	0.005	0.012	0.025
256	0.110	0.032	0.047	0.100
512	0.860	0.231	0.188	0.399
1024	6.79	1.76	0.750	1.59
2048	54.0	13.7	3.68	6.35

TABLE 6

Speedup Factor for Gaussian Elimination

Equations	2 × 2 PEs	4 × 4 PEs	8 × 8 PEs	16 × 16 PEs
64	3.3	4.9	2.2	1.0
128	3.6	10.0	4.4	2.1
256	3.8	13.3	8.9	4.2
512	3.9	14.6	17.9	8.4
1024	4.0	15.3	35.9	16.9
2048	4.0	15.6	58.4	33.8

5.7 MULTILAYER PERCEPTRON LEARNING BY BACK PROPAGATION

The multilayer perceptron is a neural network architecture that is widely used to recognize patterns in successes was NETtalk [6], a system which learned to produce natural-sounding speech from ASCII text. This section includes a brief description of a simple form of the algorithm, emphasizing the partitioning of data for MeshSP implementation. More details can be found in a number of standard texts [7].

The multilayer perceptron consists of a number of layers of units, each of which receives input from units in the layer just below, and supplies output to the layer just above. The lowest layer receives input data, rather than the output of a lower level. The activation of the j^{th} unit of level α, x^{α}_{j}, is a linear sum of the outputs of layer α-1.

$$x^{\alpha}{}_{j} = \sum_{i} y^{\alpha-1}{}_{i}\, w^{\alpha}{}_{i,j} - \theta_{j} \tag{14}$$

Here, $y^{\alpha-1}{}_{i}$ is the output of unit *i* in layer α-1, and θ_j is a threshold that may depend on the unit. The output of a unit is a sigmoidal function of its activation, going from 0 for large negative values of activation to 1 for large positive values, with a non-negative slope everywhere.

$$y^{\alpha}{}_{i} = s(x^{\alpha}{}_{i}) \tag{15}$$

The "knowledge" of the net is contained in the values of the weights $w^{\alpha}_{i,j}$. These weights are adjusted in a process of supervised learning called the "generalized delta rule." This rule adjusts each weight according to

$$\Delta w^{\alpha}{}_{i,j} = -\eta\, y^{\alpha-1}{}_{i}\, \epsilon^{\alpha}{}_{j} \tag{16}$$

Here η is a scaling constant, and ϵ^{α}_{j} is an error measurement. If the unit is an output unit, the error is the difference between the actual output and the target output.

$$\epsilon^{\alpha}{}_{j} = y^{\alpha}{}_{j} - t^{\alpha}{}_{j} \qquad \textit{output unit} \tag{17}$$

If the unit is not an output unit, the definition of error is less direct.

$$\epsilon^{\alpha}{}_{j} = s'(x^{\alpha}{}_{j}) \sum_{k} w^{\alpha+1}{}_{j,k}\, \epsilon^{\alpha+1}{}_{k} \tag{18}$$

5.7.1 MeshSP Implementation

For simplicity, assume that each layer has *N* units, where *N* is a multiple of the array dimension (8 for MeshSP-1). The weights of each layer then form an *N*×*N* array. These weights may be distributed across the array so that each slave contains an N/8×*N*/8 subarray.

Figure 5-11 illustrates the four steps in a single stage of propagation. The arrangement of input data is shown in the upper left. These may be the original input or the results of a previous stage. The first step consists of spreading these data in the row or *x*-direction so that each slave in every processor row contains contains *all* inputs associated with that row. This allows each slave to compute its *partial* contributions to

the eight outputs associated with its column on the basis of its weight subarray (lower left). For example, the slave in the first row and second column will compute contributions to outputs 8-15 with inputs 0-7. The partial sums may then be rearranged (via the permutation communication pattern) so that all eight partial sums for each output are placed within a single slave. The partial sums may then be combined and appropriately transformed or multiplied according to Equations (14) or (18), depending on whether the propagation is forward or backward. As shown in the lower right, the final output is not in the same order as the original input data, but is its transpose.

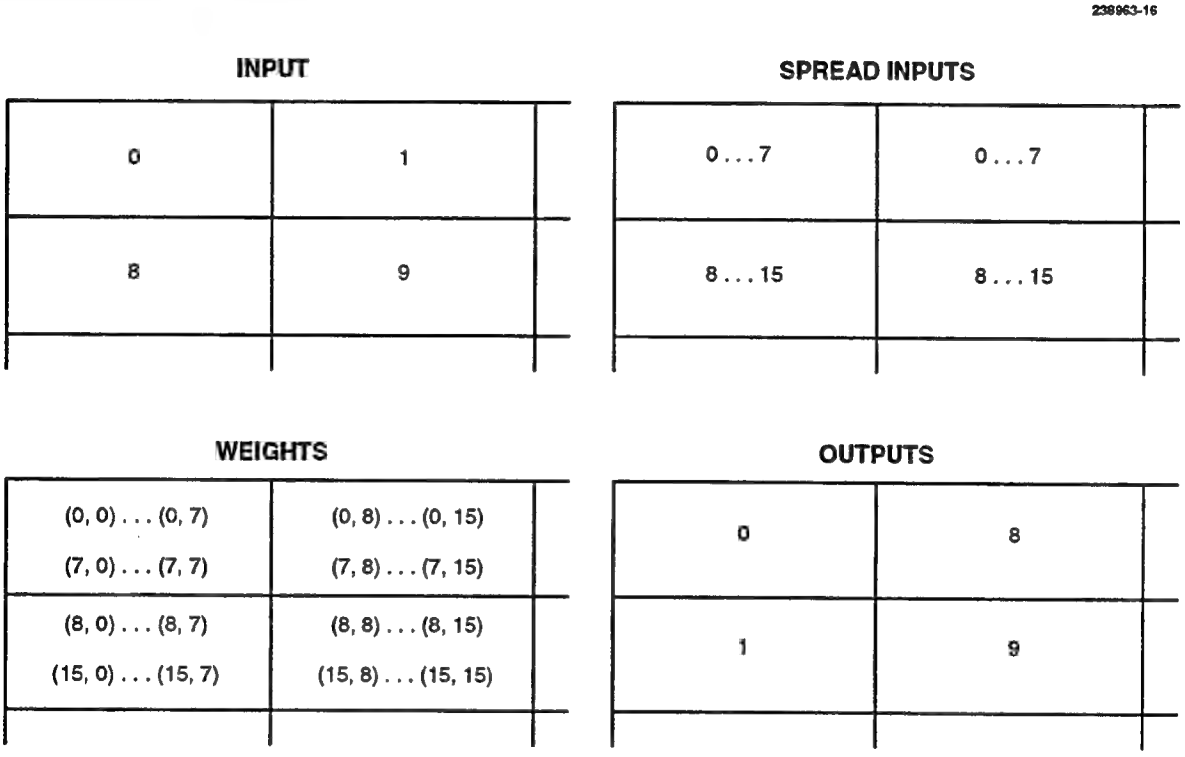

Figure 5-11. Propagation steps.

This poses no difficulty, but it does require the next stage of propagation spread these data in the y-direction (rather than x) and permute in the x-direction, rather than y. In general, at each stage the x and y communication directions must be interchanged.

5.7.2 Efficiency and Communication Cost

For data dimensions that are a multiple of the number of processors (i.e., N=64n for MeshSP-1) the MeshSP decomposition incurs no computational penalty. The communication cost is readily estimated.

Assume the processor array to be square: $AD \times AD$. Each stage of the processing requires spreading N/AD^2 inputs and permuting $(N/AD)^2$ partial sums per slave. Four cycles are required to communicate a

single 32-bit word to a nearest neighbor. Therefore, the cost of distributing N/AD^2 inputs over an entire row or column is $4N/AD$ cycles. Furthermore, the *average* path length (half the maximum path length) for the permute operation is $AD/4$ cells. Therefore, the cost of permuting the N/AD partial sums is $4(AD/4)(N/AD) = N$ cycles. Thus, the total communication cost is $4N/AD + N$ cycles. For comparison, the cost of performing the multiply accumulate operation for the partial sums is given by $(N/AD)^2$. Thus, the ratio of communication to computation is less than $AD(4 + AD)/N$. The cost of communication is less than the cost of computation whenever $N > AD(AD+4)$, e.g., $N > 96$ for $AD = 8$. Because forward and back propagation are treated symmetrically, this result holds for both directions.

5.8 TOMOGRAPHIC RECONSTRUCTION

Tomography (from the Greek *tomos*, section) refers to the cross-sectional imaging of an object. The most familiar examples are found in the technology of medical imaging, where one measures some quantity of diagnostic value at a location inside the body. Modern medical imaging techniques such as x-ray Computed Tomography (CT), Magnetic Resonance Imaging (MRI), Positron Emission Tomography (PET), and Single Photon Emission Computed Tomography (SPECT) depend on the numerical solution of a class of inverse problems. The prototypical example is the determination (reconstruction) of a two-dimensional spatial density function on a plane from a set of one-dimensional integrals at various angles. For example. the x-ray CT scan consists of measurements of the transmission of a pencil-beam, which is related to the integral of x-ray absorption density through the body on that line (see Figure 5-12). A set of integrals with a particular integration direction is known as a *projection*. Modern medical imaging systems can use projections at hundreds of different angles.

Reconstructions of functions from their integrals are used in an increasingly wide range of applications. Well-established nonmedical applications are found in the fields of radioastronomy, electron microscopy, geology, and nondestructive materials testing. The solution of the mathematical problem of reconstruction dates to a paper by Radon in 1917. The recent explosion of interest in medical applications has produced a mature collection of numerical techniques, which are described extensively in the literature. For a good introduction to computerized tomographic imaging, see the book by Kak and Slaney [8].

We will discuss an algorithm widely used in CT scanners called *filtered backprojection*. We will show that the problem can be cast naturally in a SIMD form which allows computation to proceed on the MeshSP with "complete efficiency." Complete efficiency is defined as

1. There are no computations in the parallel algorithm that would not take place in the serial version.
2. There is no overhead cost forced by interslave communications.
3. The time to reconstruct is inversely proportional to the number of PEs in the MeshSP array.

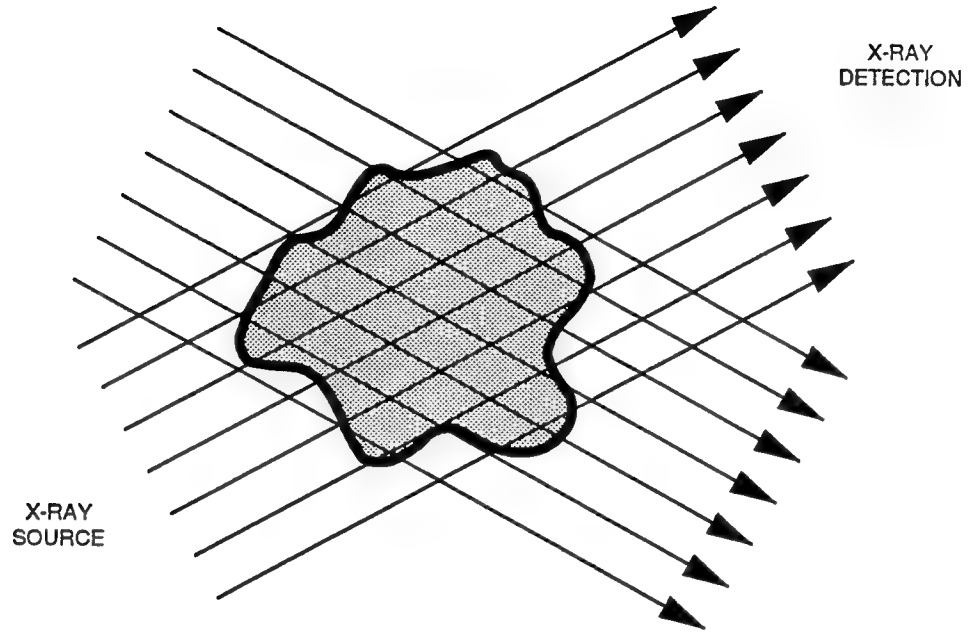

Figure 5-12. CT measurements for two projection angles.

5.8.1 An Example Problem

The geometry of our reconstruction problem is given in Figure 5-13. A density function f is defined on the (x,y)-plane within the unit circle. Outside the circle, $f = 0$. Some number of projection functions $p_\theta(t)$ are available. For every point t, $p_\theta(t)$ is a line integral through the density.

$$p_\theta(t) = \int f(t,s)\, ds \tag{19}$$

where t and s are the rotated coordinates

$$\begin{aligned} s &= x\cos\theta + y\sin\theta \\ t &= -x\sin\theta + y\cos\theta \end{aligned} \tag{20}$$

and θ is the angle between the x-axis and the s-axis (projection direction).

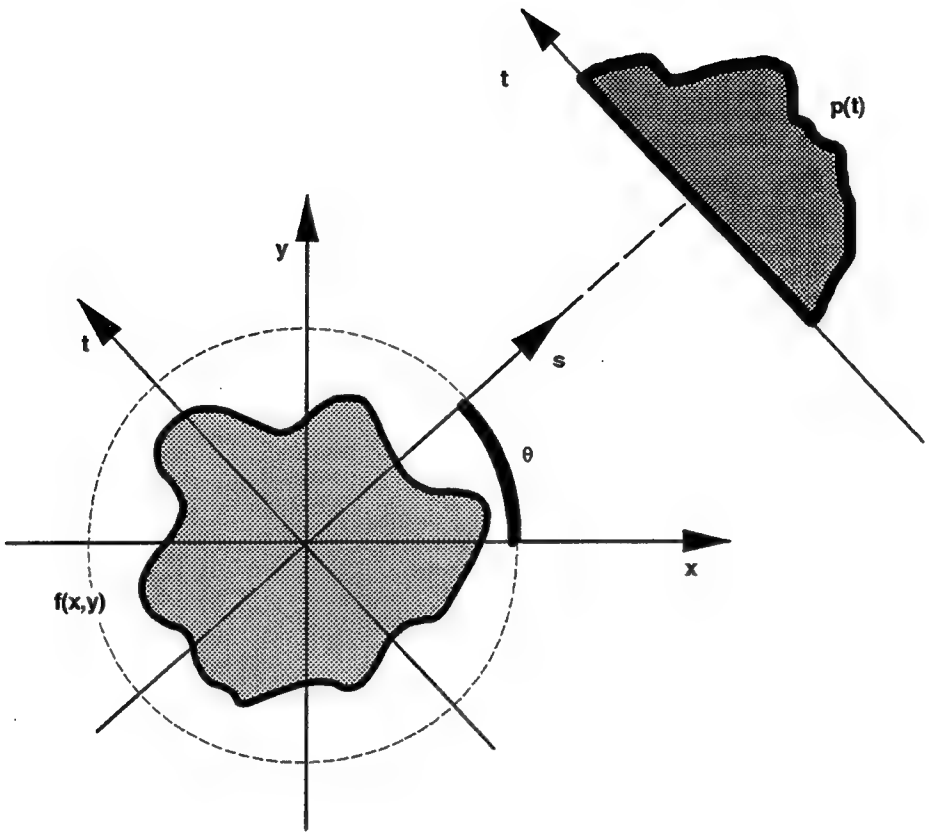

Figure 5-13. The geometry of projection.

The information contained in the continuous density function $f(x,y)$ is preserved in the projections p to an extent that depends on the sampling interval dt along t, the spatial resolution along t, and the number of projections available. The reconstruction problem is stated

Given projections $p_\theta(t)$ for several different angles, estimate $f(x,y)$.

In the x-ray CT example, p is related to the attenuation of an x-ray beam due to absorption in the body. It should be noted that there are many variations on the reconstruction problem depending on the physics of the particular situation. For example, x-ray systems are subject to an effect known as beam hardening, where

the average energy of the x-ray photons increases as the beam propagates through tissue, violating the assumption that transmitted energy flux depends on the integral of tissue density. Another variation is the fan-beam geometry used by many x-ray systems, where the integration paths making up a projection are not mutually parallel. Only the simple geometrical problem posed above is addressed because it is a starting point for a wide range of applications.

5.8.2 Filtered Backprojection

The Fourier-transformed projections p are related to the two-dimensional Fourier transform F of the density f by the Fourier slice theorem. It states that the transform of a projection

$$P(k_t) = \int p(t)\, e^{-2\pi i k_t t}\, dt \tag{21}$$

is equal to the function found from the two-dimensional transform of the density

$$F(k_x, k_y) = \iint f(x,y)\, e^{-2\pi i \vec{k}\cdot\vec{x}}\, dx\, dy \tag{22}$$

by taking values from the line in the (k_x, k_y)-plane passing through the origin and making an angle $\theta + 90°$ with the k_x-axis, as shown in Figure 5-14(a).

A reconstruction can be done by generating the two-dimensional field $F(k_x,k_y)$ via interpolation from the transformed projection data $P_\theta(k_t)$, where the projections p have been sampled at a sufficiently fine spatial resolution dt and angular spacing dθ. The density function f is then obtained from an inverse two-dimensional transform of F.

Although the preceding prescription seems very straightforward, the filtered backprojection algorithm is more often employed in practice. Consider a set of projections at equally spaced angles covering the interval $0° \le \theta < 180°$. This suffices for our simple integrals; though real applications often benefit from 360° coverage. Consider projection $p_\theta(t)$. If one constructs a density function g such that

$$g(x,y) = c\, p_\theta(\, -x \sin\theta + y \cos\theta\,) \tag{23}$$

then one has "backprojected" p_θ to the (x,y)-plane. The constant c can be adjusted to make the θ-projection of g equal to p_θ. Backprojecting all the p's will not reconstruct the original density. This is understood with the help of the Fourier slice theorem: the projections $P(k)$ contribute too heavily at the low spatial frequencies by a factor of $1/|k|$. Each projection contributes a band [as in Figure 5-14(a)] with a width determined by the

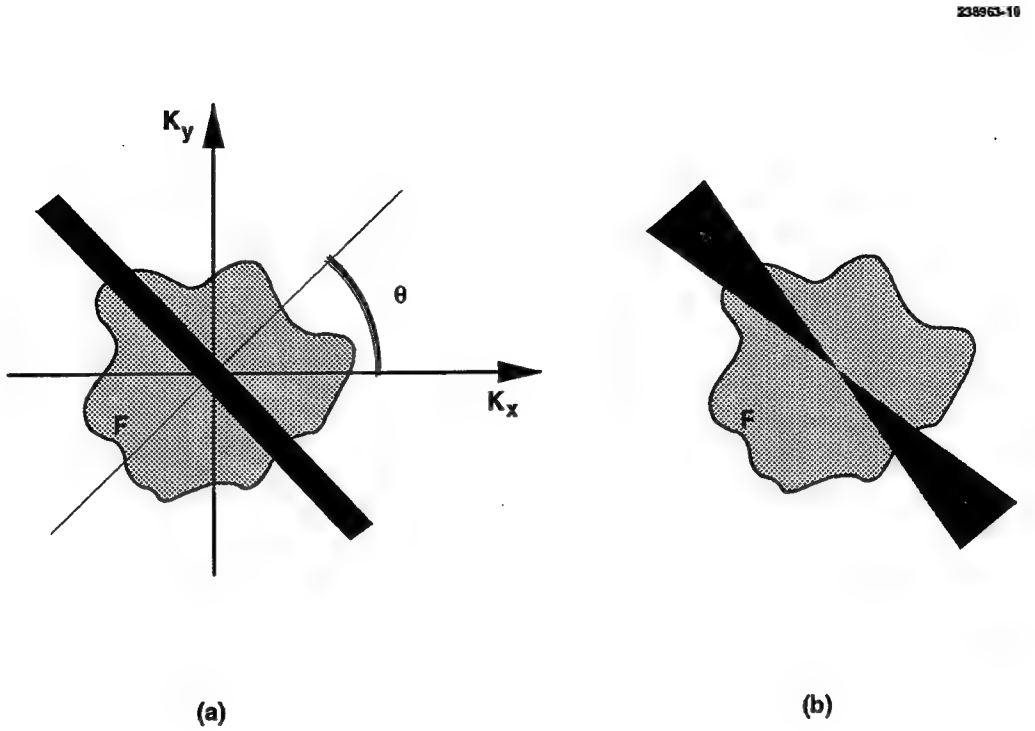

Figure 5-14. A slice of Fourier space.

extent of the $p(t)$ measurement. Ideally, one would like to sum pie-slice-shaped contributions to Fourier space, as suggested in Figure 5-14(b). This situation can be approximated by filtering (weighting) the $P(k)$ with the function $|k|$. In practice one also rolls-off the high frequencies in $P(k)$ to an extent that depends on the angular sample spacing and measurement noise. The desired density is reconstructed by backprojecting the complete set of filtered projections $p'(t)$ in (x,y)-space. The advantages of filtered backprojection for medical imaging system are

1. Filtering and backprojection can begin for each projection as soon as it becomes available. Reconstruction thus proceeds in parallel with data collection. Two-dimensional Fourier reconstruction requires all the data to be available at the start.

2. To control patient exposure, one can add projections incrementally until the desired information is evident.

3. Properly interpolating projection data in two-dimensional Fourier space is more complicated than the linear interpolation used in backprojection.

5.8.3 A MeshSP Implementation

At first sight, one might assume the backprojection geometry to be ill suited to mesh SIMD processing. In fact, the reconstruction problem can be implemented on the MeshSP in a very straightforward way and with nearly complete efficiency. The key is to map the desired density function to the two-dimensional slave array and pass each projection function through all the slaves via the interprocessor communication network. The 8×8 algorithm proceeds as follows.

1. A TCB is constructed that will transmit one filtered projection $p'(t)$ to a neighboring slave. A one-dimensional ring of connections is established between the 64 slaves, so that a projection function may be passed through all the slaves in bucket-brigade fashion by calling the CM() function 63 times. A suitable communication pattern on the toroidally connected MeshSP array is shown in Figure 5-15.

2. Sixty-four projections are downloaded to the slaves via the SIO system. Each slave receives one projection. The projections may represent any angle. Reconstruction from these angles may proceed while additional data are being collected. The complete reconstruction may consist of many sets of 64 projections. If the total number of projections is not a multiple of 64, the number is augmented with zeroed projections.

3. Each slave filters its projection data. We implement the filtering as a weighting of the transformed $P(k)$. The projections p are zero-padded to avoid wrap-around aliasing. An inverse transform provides the filtered projections p' in real-space.

4. Each slave initiates communication of the filtered projection to the next slave. This is carried out concurrently with the backprojection operation. For a square array of width N samples, each filtered projection requires N*sqrt(2) samples to permit backprojection to all parts of the final density. The filtered projection p_θ' is accompanied by the angle θ (to identify the projection) and the numbers sinθ and cosθ (to save computation).

5. Each slave backprojects to the area assigned to it. This is done with a loop over the final points $f(x,y)$, a coordinate transform to t, and a linear interpolation between projection data cells. Because each slave is concerned only with the density cells assigned to it, there is no wasted computation.

6. The concurrent combination of communication and backprojection repeats for subsequent projections.

7. While the 64 backprojections are in progress, the next set of 64 projections can be downloaded in the background via the SIO system.

8. After all projections have been processed the reconstructed density f is uploaded to the host via the SIO.

The assertion that this MeshSP algorithm proceeds efficiently depends on the ability to "hide" the data transfers behind the backprojection computation. The transfer of the projection to the next slave is always nearest neighbor and requires $4*N*\text{sqrt}(2)$ cycles. The backprojection operation is performed for $(N/8)^2$ cells, with an estimated ten cycles per cell required for the coordinate rotation and linear interpolation. For $N = 256$, the communication time is 15% of the computation time.

5.8.4 MeshSP Timing

An estimate of the real time to reconstruct is made with the following definitions.

N^2	is the number of cells in the output density.
N	is the number of nonzero cells in a projection.
$2N$	is the number of cells in a zero-padded projection.
$1.42N$	is the number of cells in a filtered projection.
S^2	is the number of MeshSP slaves in the square array.
M	is the number of projections (a multiple of S^2).

Timing estimates are given below for various components of the algorithm for the following case: $N = 256$, $S = 8$, and $M = 1024$. A 40-MHz processor clock is assumed.

Projection download. Time is allocated to download the first S^2 projections to the slave array via the SIO. Subsequent downloads happen in the background during computation. Each MeshSP slave column receives NS data points at 5×10^6 bytes/sec. The required time is $8NS\times10^{-7}$ sec = 2 ms.

Roundtrip FFTs. Each slave transforms the zero-padded projection of a real array of $2N$ samples for each new set of projections (M/S^2 sets). The one-way transform time is approximately $1.0(2N)\log_2(2N)$ cycles. The required time is $(NM/S^2)\log_2(2N)10^{-6}$ sec = 37 ms.

Filtering. Weighting by the precomputed k-space filter requires $(M/S^2)2N$ multiplications and stores. The required time is $N(M/S^2)10^{-6}$ sec = 4 ms.

Backprojection. This requires ten cycles per density cell per projection. There are $(N/S)^2$ cells per slave. The required time is $0.25M(N/S)^2 10^{-6}$ sec = 262 ms.

Density Upload. Each MeshSP-1 column must upload SN^2 floats over the bit-serial SIO at 5 Mbytes/sec. The required time is $N^2/S\ 4\times2\times10^{-7}$ sec = 7 ms.

The total reconstruction time of a 256×256 cell density function from 1024 projections is 312 ms. Other reconstruction times are listed in Table 7 for different image sizes and numbers of projections.

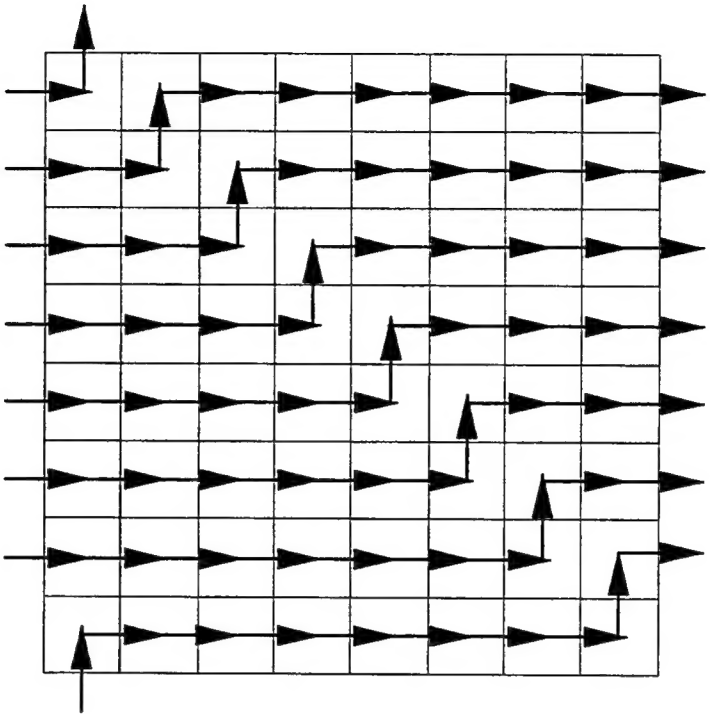

Figure 5-15. A one-dimensional ring on a two-dimensional toroidal processor array.

TABLE 7

Time for Reconstruction (sec)

	PROJECTIONS				
Image Size	128	256	512	1024	2048
128 x 128	0.013	0.023	0.044	0.086	0.17
256 x 256	0.046	0.084	0.17	0.31	0.61
512 x 512	0.17	0.31	0.60	1.2	2.3
1024 x 1024	0.66	1.2	2.3	4.5	8.9
2048 x 2048	2.6	4.7	9.0	17.0	35.0

REFERENCES

1. I.H. Gilbert, "The synchronous processor," *Lincoln Lab. J.* **1**, 1 (1988).

2. *ADSP-21060 SHARC, Preliminary User's Manual,* Analog Devices, Inc. (September 1993).

3. K.A. Gallivan et al., "Parallel algorithms for dense linear algebra computations," *SIAM Rev.* 31, 1, 54–135 (1990).

4. D.P. Bertsekas and J.N. Tsitsiklis, *Parallel Computation,* Englewood Cliffs, NJ: Prentice Hall (1984).

5. W.H. Press et al., *Numerical Recipes in C,* ed. 2, New York, NY: Cambridge University Press (1992). See pages 36 and 55.

6. T. Sejnowski and C.R. Rosenberg, *NETtalk: A parallel network that learns to read aloud,* Dept. of Electrical Engineering and Computer Science, Johns Hopkins University, Baltimore, MD, Technical Report 86-01 (1986).

7. D. Rumelhart, J. McClelland, and the PDP Research Group, *Parallel Distributed Processing,* Vol. 1, Cambridge, MA: MIT Press (1986). See pages 322–328.

8. A.C. Kak and M. Slaney, *Principles of Computerized Tomographic Imaging,* New York, NY: IEEE Press (1988).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	14 December 1994	Technical Report

4. TITLE AND SUBTITLE

The Mesh Synchronous Processor MeshSP™

5. FUNDING NUMBERS

C — F19628-95-C-0002

6. AUTHOR(S)

Ira H. Gilbert and William S. Farmer

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Lincoln Laboratory, MIT
P.O. Box 73
Lexington, MA 02173-9108

8. PERFORMING ORGANIZATION REPORT NUMBER

TR-1004

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

U.S. Air Force
Washington, D.C. 20330

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

ESC-TR-94-087

11. SUPPLEMENTARY NOTES

MeshSP is a trademark of the Massachusetts Institute of Technology.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

The Mesh Synchronous Processor (MeshSP) is a parallel computer architecture, primarily SIMD, combining high throughput with modest size, weight, power, and cost. Each MeshSP processor node consists of a single DSP chip: the ADSP-21060 (SHARC) chip of Analog Devices Inc. The MeshSP-1 processor, a hardware realization of the MeshSP, incorporates an 8 × 8 array of ADSP-21060 chips, providing a peak throughput of 7 GFlops. The processor is programmed in ANSI C or C++ with no parallel extensions. A combination of on-chip DMA hardware and system software makes data I/O and interprocessor communication uniquely simple. The MeshSP is easily programmed to solve a wide variety of computationally demanding signal-processing problems. A functional simulator enables MeshSP algorithms to be coded and tested on a personal computer. Some example applications are described.

14. SUBJECT TERMS

computer architecture
digital signal processors
single instruction-stream multiple data stream (SIMD)
parallel processors
simulators
linear algbra
neural net simulations
parallel algorithms
fourier transforms
tomography

15. NUMBER OF PAGES
104

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Same as Report